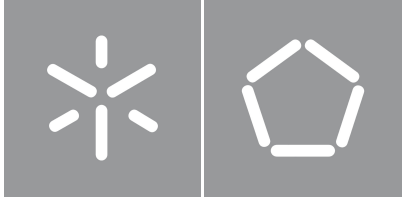


Universidade do Minho
Escola de Engenharia

Maria José Costa Ramos

Injeção de faltas reprodutível em sistemas de armazenamento local

Dezembro, 2023



Universidade do Minho
Escola de Engenharia

Maria José Costa Ramos

Injeção de faltas reprodutível em sistemas de armazenamento local

Dissertação de Mestrado
Mestrado em Engenharia Informática

Trabalho realizado sob a orientação de
João Tiago Medeiros Paulo (Orientador)
Ricardo Gonçalves Macedo (Co-orientador)
Tânia da Conceição Araújo Esteves (Supervisora no Centro de Investigação)

Dezembro, 2023

Direitos de Autor e Condições de Utilização do Trabalho por Terceiros

Este é um trabalho académico que pode ser utilizado por terceiros desde que respeitadas as regras e boas práticas internacionalmente aceites, no que concerne aos direitos de autor e direitos conexos.
Assim, o presente trabalho pode ser utilizado nos termos previstos na licença abaixo indicada.
Caso o utilizador necessite de permissão para poder fazer um uso do trabalho em condições não previstas no licenciamento indicado, deverá contactar o autor, através do RepositóriUM da Universidade do Minho.

Licença concedida aos utilizadores deste trabalho:



CC BY 4.0

https://creativecommons.org/licenses/by/4.0/

Agradecimentos

Um agradecimento aos meus orientadores, o João Paulo, o Ricardo Macedo e a Tânia Esteves, que possibilitaram e ajudaram no desenvolvimento desta dissertação.

Este trabalho foi financiado por Fundos Nacionais através da agência de financiamento portuguesa, FCT - Fundação para a Ciência e a Tecnologia, no âmbito dos projetos LA/P/0063/2020 e UIDP/50014/2020, e pela Componente 5 – Capitalização e Inovação Empresarial, integrada na Dimensão Resiliência do Plano de Recuperação e Resiliência no âmbito do Mecanismo de Recuperação e Resiliência (MRR) da União Europeia (UE), enquadrado no Próximo Geração EU, para o período 2021 - 2026, no âmbito do projeto ATE, com referência 56.

Declaração de Integridade

Declaro ter atuado com integridade na elaboração do presente trabalho académico e confirmo que não recorri à prática de plágio nem a qualquer forma de utilização indevida ou falsificação de informações ou resultados em nenhuma das etapas conducente à sua elaboração.

Mais declaro que conheço e que respeitei o Código de Conduta Ética da Universidade do Minho.

Maria Ramos

Universidade do Minho, Braga, dezembro 2023

Maria José Costa Ramos

Resumo

Injeção de faltas reprodutível em sistemas de armazenamento local

A era digital trouxe a incorporação dos computadores em vários setores do nosso quotidiano. Tem-se verificado, ao longo dos anos, uma crescente dependência na tecnologia, o que se traduz na necessidade de sistemas cada vez mais resilientes, rápidos, seguros e disponíveis.

Consequentemente, a complexidade dos sistemas tem vindo a crescer, o que leva à implementação de mecanismos de persistência e atualização de dados mais difíceis de testar. Para elevar o nível de dificuldade, os diversos sistemas de ficheiros possuem particularidades que afetam de forma significativa estes mecanismos e a consistência das aplicações após a ocorrência de falhas de energia e dos sistemas operativos.

Evitar a perda total ou parcial dos dados deve ser um ponto fulcral para sistemas de armazenamento com fortes garantias de durabilidade e coerência. Na literatura, os sistemas capazes de voltar a um estado coerente após uma falha de energia são chamados de *crash-consistent*. Muito do trabalho relacionado foca-se na *crash consistency* dos sistemas de ficheiros e, quando se foca nas aplicações, verifica-se a falta de ferramentas para reprodução de *bugs*. Quando um utilizador reporta problemas encontrados numa dada aplicação, é útil para as equipas de desenvolvimento terem uma ferramenta que rapidamente reproduz o *bug* e ajude na sua correção.

Esta dissertação propõe o LazyFS+, um sistema de ficheiros que simula a perda total e parcial de dados através da injeção de faltas baseada em *software*. A nossa solução possui uma cache interna, o que permite uma gestão determinística dos dados das aplicações e a injeção de faltas reprodutível. Um dos seus pontos fortes é o facto de imitar comportamentos que alguns sistemas de ficheiros apresentam, como a reordenação de escritas, sem se prender a uma implementação específica. Para além disso, facilita a análise das operações executadas pela aplicação e permite obter informação sobre os dados não sincronizados.

O LazyFS+ provou-se útil através da reprodução de *bugs* já conhecidos e da identificação desses mesmos *bugs* em versões antigas das aplicações, nunca antes reportados. Para além disso, foi possível encontrar novos *bugs*, quer em versões antigas, quer em versões mais recentes das aplicações. Estes últimos foram reportados com todos os passos de reprodução realizados com o LazyFS+. Como resultado, atualmente este está a ser integrado nos testes do sistema de armazenamento chave-valor etcd.

Palavras-chave: *Crash consistency*, Injeção de faltas, Reprodutibilidade

Abstract

Reproducible fault injection in local storage systems

The digital era has brought the incorporation of computers into various sectors of our daily lives. Over the years, there has been an increasing dependence on technology, which translates into the need for increasingly resilient, fast, secure and available systems.

Consequently, the complexity of systems has been growing, which leads to the implementation of data persistence and update mechanisms that are more difficult to test. To increase the level of difficulty, the different file systems have particularities that significantly affect these mechanisms and the consistency of applications after power and operating system failures.

Avoiding total or partial loss of data must be a key point for storage systems with strong guarantees of durability and consistency. In the literature, systems capable of returning to a consistent state after a power failure are called crash consistent. Much of the related work focuses on the crash consistency of file systems, and when it focuses on applications, there is a lack of tools for reproducing bugs. When a user reports issues encountered in a particular application, it is useful for development teams to have a tool that quickly reproduces the bug and helps with its fix.

This dissertation proposes LazyFS+, a file system that simulates total and partial data loss through software-based fault injection. Our solution has an internal cache, which allows deterministic management of application data and reproducible fault injection. One of its strengths is the fact that it imitates behaviors that some file systems present, such as the reordering of writings, without being tied to a specific implementation. Furthermore, it facilitates the analysis of the operations carried out by the application and allows to obtain information about unsynchronized data.

LazyFS+ proved useful by reproducing already known bugs and identifying these same bugs in older versions of applications, never before reported. Furthermore, it was possible to find new bugs, both in old and newer versions of the applications. The latter were reported with all reproduction steps performed with LazyFS+. As a result, this is currently being integrated into testing of the etcd key-value store.

Keywords: Crash consistency, Fault Injection, Reproducibility

Conteúdo

1 Introdução **1**

1.1 Problema . . . 2

1.2 Objetivos e Contribuições . . . 4

1.2.1 Resultados . . . 4

1.3 Estrutura do Documento . . . 5

2 Estado da Arte **6**

2.1 Faltas . . . 6

2.1.1 Classificação . . . 7

2.1.2 Confiabilidade . . . 8

2.1.3 Tolerância a faltas . . . 9

2.1.4 Injeção de faltas . . . 9

2.1.5 Reprodutibilidade . . . 12

2.2 Sistemas de armazenamento . . . 13

2.2.1 Fluxo de dados . . . 15

2.2.2 Persistência de dados . . . 16

2.2.3 *Crash consistency* . . . 17

2.3 Trabalho relacionado . . . 20

2.3.1 *Crash consistency* em sistemas de ficheiros . . . 20

2.3.2 Exploração de estados de falha em aplicações . . . 21

2.3.3 Injetores de faltas . . . 24

2.3.4 Discussão . . . 27

3 Motivação **30**

3.1 Metodologia . . . 30

3.2 Falta de *fsync* . . . 31

3.3 Reordenação de escritas . . . 33
3.4 Tamanho das escritas . . . 34
3.5 Discussão . . . 35

4 LazyFS+ 37
4.1 Modelo de faltas . . . 38
4.2 Fluxo de operações . . . 39
4.2.1 Leitura . . . 41
4.2.2 Escrita . . . 41
4.2.3 Sincronização . . . 41
4.2.4 Faltas injetadas na Interface de Controlo . . . 42
4.2.5 Faltas configuradas . . . 42
4.3 Arquitetura . . . 43
4.3.1 Cache de Páginas . . . 43
4.3.2 *Backend* de Armazenamento . . . 44
4.3.3 Interface de Controlo . . . 45
4.3.4 Gestor de Escritas . . . 46
4.4 Implementação . . . 47
4.5 Configuração . . . 48
4.6 Comandos da Interface de Controlo . . . 51
4.7 *Output* . . . 51

5 Avaliação 56
5.1 Metodologia . . . 56
5.2 Estudo . . . 58
5.2.1 Falta de perda total de dados em cache . . . 58
5.2.2 Faltas de reordenação . . . 68
5.2.3 Faltas de atomicidade . . . 74
5.2.4 Validação e mecanismos de recuperação . . . 78
5.3 Discussão . . . 81

6 Conclusão 84
6.1 Trabalho futuro . . . 85

Lista de Figuras

1 Arquitetura geral de um sistema de injeção de faltas. 10

2 Pilha de armazenamento tradicional. 14

3 Reordenação de escritas. 33

4 Arquitetura do LazyFS+ e fluxo de operações. Os componentes a violeta foram modificados e/ou adicionados à arquitetura base do LazyFS. 40

5 Estrutura da Cache de Páginas. Neste exemplo, o *inode* possui quatro blocos de dados e cada página é composta por dois desses blocos. 44

6 Exemplo de grafo produzido pelo *parser* a partir de um log do LazyFS+. 54

7 Sequência de chamadas ao sistema do v2.3.0 etcd no momento de inicialização, onde nenhuma falta foi aplicada. Imagem produzida pelo *parser* do LazyFS+. 64

8 Sequência de chamadas ao sistema executadas pelo etcd v3.4.25 no momento de inicialização, sem que nenhuma falta seja aplicada. Imagem produzida pelo *parser* do LazyFS+. 66

9 Sequência de chamadas ao sistema executadas pelo ZooKeeper v3.7.1 quando o cliente se conecta a um servidor, sem que nenhuma falta tenha sido injetada. Imagem produzida pelo *parser* do LazyFS+. 69

10 Parte da sequência de chamadas ao sistema realizadas pelo ZooKeeper v3.7.1 quando a carga de trabalho é executada, sem a injeção de faltas. Imagem produzida pelo *parser* do LazyFS+. 71

11 Conjunto de chamadas ao sistema invocadas na execução de uma carga de trabalho no LevelDB 1.15, sem a injeção de faltas. Imagem produzida pelo *parser* do LazyFS+. . . . 73

12 Chamadas ao sistemas executadas pelo Redis v7.2.3 quando é executada uma carga de trabalho, sem a injeção de faltas. Imagem produzida pelo *parser* do LazyFS+. 79

Lista de Tabelas

1 Resumo do trabalho relacionado. A coluna "sistemas testados"indica em que tipo de sistemas as ferramentas foram efetivamente testadas. As ferramentas em que não foram encontrados dados acerca de sistemas reais testados têm uma indicação de "não aplicável". Uma injeção ser simulada indica que as faltas não são injetadas em tempo de execução, mas os sistemas são executados, enquanto que a injeção ser estática significa que os sistemas não são executados. 28

2 Sumário dos testes realizados com o LazyFS+. Cada teste possui uma etiqueta que tem uma correspondência com o texto da Secção 5. As linhas a cor de laranja referem-se a testes de validação, ao contrários dos outros, que constituem problemas encontrados nas aplicações. 59

Acrónimos

ACID Atomicidade, Consistência, Isolamento e Durabilidade. 21, 22, 57

AMP *Abstract Persistence Model*. 22

E/S Entrada/Saída. 13, 15–17, 20, 23, 24

FUSE *File System in User Space*. 24–27, 47

HDD *Hard Drive Disk*. 1, 13, 15

IPC *Interprocess Communication*. 11

LRU *Least Recently Used*. 44

POSIX *Portable Operating System Interface*. 1, 4, 15, 17, 27, 37, 47

RAM *Random Access Memory*. 15, 56

RPC *Remote Procedure Call*. 11

SF Sistema de Ficheiros. 1–4, 9, 14–26, 30, 37–39, 44, 45, 47, 56, 58, 65, 69, 70, 74, 84, 85

SO Sistema Operativo. 1, 2, 11, 13, 16, 17, 19, 23, 26, 44, 56, 84

SSD *Solid-state Storage Device*. 2, 13

UCP Unidade Central de Processamento. 56

UPS *Uninterruptible Power Supply*. 1

WAL *Write-Ahead Logging*. 19, 24, 31, 35, 63, 65, 71, 72, 74, 75, 80, 81

Capítulo 1
Introdução

Muitas aplicações, como bases de dados, sistemas de armazenamento chave-valor, serviços de coordenação, entre outros, requerem a durabilidade e consistência dos seus dados na presença de faltas como falhas de energia ou falhas do Sistema Operativo (SO). Todos os dados que se encontravam apenas em memória volátil, até ao momento da ocorrência destas falhas, são perdidos. Em ambos os casos mencionados, as aplicações não têm a oportunidade de terminar corretamente todos os processos e persistir os dados que se encontravam temporariamente armazenados em memória.

Apesar do papel importante desempenhado pelos dados, há vários casos em que os sistemas não foram capazes de tolerar faltas, que, consequentemente, resultaram na indisponibilidade ou perda de dados. Em 2009, uma falha no servidor do Microsoft Sidekick levou à perda de uma grande quantidade de dados como contactos e fotos de cerca de 800 mil utilizadores [28]. Já em 2015, um relâmpago levou à indisponibilidade dos serviços disponibilizados pela Google Cloud durante quatro dias consecutivos [98]. Foi estimado que cerca de 5% dos discos da zona afetada apresentaram erros de leitura e escrita e uma pequena percentagem destes permaneceram afetados, sem possibilidade de recuperação. Estes problemas podem afetar a longo prazo uma empresa, visto que podem resultar na perda de confiança por parte dos utilizadores.

Alguns centros de dados possuem geradores e/ou sistemas conhecidos como *Uninterruptible Power Supply* (UPS), que funcionam como reservas de energia em caso da ocorrência de falhas de energia, de forma a evitar ou minimizar a ocorrência de faltas. Porém, nem todas as organizações são capazes de suportar os custos iniciais e de manutenção deste tipo de tecnologias [43, 96], para além de outras desvantagens, tais como a pegada ambiental, no caso de geradores e, no caso dos sistemas UPS, o facto de não serem destinados a grandes aplicações [64].

Numa pilha de armazenamento tradicional baseada em *Portable Operating System Interface* (POSIX), é possível tornar os dados duráveis através da chamada ao sistema `fsync`. Esta operação é utilizada para fazer sincronização explícita dos dados em memória em diferentes camadas da pilha de armazenamento (*e.g.*, Sistema de Ficheiros (SF), *block device*) para o dispositivo de armazenamento (*e.g.*, *Hard Drive Disk*

(HDD), *Solid-state Storage Device* (SSD)).

O `fsync` é crucial para a prevenção da perda de dados dos utilizadores, mas também para garantir a consistência das aplicações em caso de falhas de energia ou dos sistemas. O problema de *crash consistency* tem sido exposto em alguns estudos [33, 39, 54, 56, 60], onde é demonstrado que muitas aplicações e SFs podem perder ou corromper dados na sequência de falhas de energia ou dos sistemas, resultando em funcionamento incorreto ou até indisponibilidade. Idealmente, os dados das aplicações seriam sincronizados com a máxima frequência possível, no entanto, o `fsync` introduz uma latência significativa às aplicações. Esta é a principal razão para a sua utilização ser evitada ao máximo.

Verifica-se que, por vezes, os próprios programadores fazem suposições erradas de como os SOs, SFs e *block devices* persistem os dados. Para evitar realizar um `fsync` a cada escrita, as aplicações realizam-no após um conjunto de escritas, ou, num único `write`, incluem uma maior quantidade de dados. Quando uma falha ocorre, os dados podem ser perdidos na totalidade ou parcialmente e fora de ordem. Isto ocorre porque as camadas na pilha de armazenamento podem reordenar e persistir escritas [39, 42]. Quando as aplicações não estão desenhadas para lidar com estes cenários, podem perder dados importantes e ficar corrompidas, muitas vezes até num estado irrecuperável (*e.g.*, os metadados de uma base de dados, necessários para a inicialização, corrompidos e irrecuperáveis devido a uma escrita parcial).

As aplicações devem, portanto, desenhar protocolos de atualização de dados cuidadosamente, de modo a garantir a durabilidade e consistência dos dados, mas sem afetar de forma dramática o desempenho. Perante este cenário, torna-se relevante desenvolver técnicas capazes de verificar se as aplicações estão a fazer o suficiente. Estudos recentes [55, 56] evidenciam que muitos sistemas não submetem os seus mecanismos de tolerância a faltas a testes rigorosos, indicando assim a necessidade urgente de consciencialização sobre a importância destes testes, bem como da disponibilidade de ferramentas suficientemente robustas para esse propósito.

1.1 Problema

Dada a relevância da preservação dos dados, torna-se importante testar os mecanismos de persistência de dados implementados pelas aplicações. Para a simulação de falhas de energia, muitas vezes é sugerido realizar algum tipo de operação e, passado um determinado período de tempo, desligar a ligação de energia do computador[1]. No entanto, para além de não ser prático, esta forma de simular a falha de

[1] *Possible bug: Missing a fsync() on the ".log"file before compaction* - https://github.com/Lendfating/leveldb/issues/187

energia não é suficiente para casos em que a janela temporal para a injeção da falta é pequena. Adicionalmente, esta forma de injetar faltas possui um nível muito baixo de reprodutibilidade, já que dificilmente se consegue cortar a energia sempre no mesmo instante temporal.

Por vezes, para reproduzir certos problemas, é necessário, para além de cortar a fonte de alimentação do computador, utilizar SFs específicos e fazer alterações ao código fonte das aplicações para introduzir atrasos em certas operações. Todo este processo é temporalmente dispendioso e bastante propenso a erros. Se, por exemplo, um *bug* ocorre por determinadas escritas serem reordenadas, pode ser que num segundo teste, essas escritas sejam persistidas por ordem. É necessário que as várias condições de teste se mantenham as mesmas e basta um processo ter sido executado, por exemplo, em segundo plano, sem o utilizador se aperceber, que a memória já não terá a mesma composição do que no teste anterior.

A reprodutibilidade de um teste é importante para compreender as potenciais causas de um dado problema e a forma como este se manifesta. A correção de um *bug* pode não ser simples, portanto a possibilidade de o reproduzir ajuda nesta tarefa.

Várias das ferramentas existentes para encontrar vulnerabilidades nos sistemas focam-se bastante na parte exploratória, ou seja, em encontrar problemas desconhecidos. Poucas são aquelas que, dado um problema conhecido, permitem a sua reprodução em pouco tempo, o que muitas vezes é útil no caso de, por exemplo, um utilizador reportar um dado problema. Em vários repositórios de aplicações, as equipas de desenvolvimento das aplicações solicitam aos utilizadores para darem todos os detalhes possíveis de como é possível reproduzir o *bug* ocorrido e, por vezes, os problemas não são corrigidos pela falta de meios de reprodução [2].

Em suma, verifica-se que existe a necessidade de desenvolver uma ferramenta que seja agnóstica às aplicações e às várias especificações de SFs existentes. Por um lado, não deve exigir a alteração ou compreensão do código fonte das aplicações e, por outro, deve permitir testar as aplicações tendo em conta propriedades não intuitivas dos SFs, sem necessitar de executar a aplicação sobre os sistemas que apresentam essas propriedades. Esta ferramenta deve também permitir o controlo total sobre os dados que são ou não persistidos, controlo sobre os comportamentos dos SFs e a simulação de falhas sempre no mesmo ponto do programa, de modo a permitir a reprodução de problemas.

[2] *ETCD data gets corrupted with error "read wal error (walpb: crc mismatch)"* - https://github.com/etcd-io/etcd/issues/6191#issuecomment-240268979

1.2 Objetivos e Contribuições

Um dos objetivos desta dissertação é disponibilizar uma forma determinística de simular a perda total dos dados das aplicações, bem como a perda parcial, que resulta do comportamentos dos SFs. Deste modo, esta dissertação apresenta o LazyFS+, um SF que permite simular falhas de energia, através da perda total e parcial de dados que não foram explicitamente *fsynced* pelas aplicações.

Esta dissertação tem também como objetivo reproduzir *bugs* encontrados em sistemas de armazenamento. Existe alguma variedade de ferramentas que testam exaustivamente os sistemas em busca de problemas, mas verifica-se que existe falta de ferramentas que permitam reproduzir *bugs* encontrados por utilizadores e que permitam entender a sua causa e a sua correção. O LazyFS+, por possuir uma cache própria, é capaz de controlar os dados das aplicações e, disponibilizando faltas que podem ser injetadas sempre no mesmo ponto do programa, permite reproduzir os mesmos cenários. Para além disso, o LazyFS+ é genérico, pois permite testar diferentes aplicações, tendo como único requisito que estas sejam compatíveis com a interface POSIX.

Em adição à reprodutibilidade, é importante disponibilizar informação que ajude na tarefa de correção dos *bugs*. Deste modo, para além de o LazyFS+ indicar os blocos de dados em risco de serem perdidos, foi desenvolvido, no âmbito desta dissertação, um *parser* que transforma os logs produzidos pelo LazyFS+. Estes logs são tornados mais legíveis e ainda é possível identificar escritas de tamanho superior a páginas da cache. Isto contribui para perceber melhor o fluxo de dados das aplicações, as causas dos problemas encontrados e, ainda, identificar pontos de vulnerabilidade.

Por fim, o LazyFS+ pretende contribuir para a procura de novos *bugs* em sistemas de armazenamento. Com o LazyFS+ foi possível encontrar *bugs* desconhecidos, quer em versões antigas de certas aplicações, quer em versões mais recentes. Um dos problemas encontrados foi ao nível do etcd, um sistema de armazenamento chave-valor, e, atualmente, o LazyFS+ está ser integrado nos testes internos desta aplicação [3].

1.2.1 Resultados

O trabalhado realizado nesta dissertação teve como resultados:

- Disponibilização do LazyFS+ num repositório público [4].

[3] *Etcd robustness tests utilize new LazyFS feature to simulate power failure* - https://github.com/etcd-io/etcd/issues/16597

[4] Repositório do GitHub - https://github.com/dsrhaslab/lazyfs

- Reprodução de múltiplos *bugs* identificados por utilizadores e reportados em artigos do trabalho relacionado. Além disso, foram encontrados 9 novos problemas. Destes, 7 pertencem a versões recentes de diversos sistemas de armazenamento (ZooKeeper, PebblesDB e etcd) e foram reportados às equipas de desenvolvimento.

- Integração do LazyFS+ em testes no etcd, quer com objetivo de encontrar problemas, quer para perceber estudar soluções dos mesmos.

1.3 Estrutura do Documento

No Capítulo 2 são abordados alguns conceitos relacionados com faltas e é analisado o fluxo de dados numa pilha de armazenamento tradicional. O capítulo termina com uma análise do trabalho relacionado, que se direciona ao estudo da durabilidade e da consistência dos dados. De seguida, no Capítulo 3, são apresentados alguns casos de estudo motivadores do desenvolvimento da solução apresentada, o LazyFS+. No Capítulo 4 é apresentado o desenho, arquitetura e implementação do LazyFS+. Já no Capítulo 5 é apresentado um conjunto de problemas reproduzidos com a solução desenvolvida, bem como alguns casos de validação. Por fim, o Capítulo 6 conclui a dissertação com o resumo do trabalho realizado e aborda tópicos que podem ser explorados no futuro.

Capítulo 2
Estado da Arte

Neste capítulo serão exploradas e consolidadas algumas noção básicas relacionadas com faltas, com o intuito de enquadrar e contextualizar o problema. Será também aprofundada a forma como ocorre o fluxo de dados num sistema, de forma a elucidar sobre o funcionamento e a necessidade de mecanismos de persistência de dados.

Para além disso, será conduzido um estudo sobre o tipo de ferramentas de injeção de faltas que existem na atualidade. Deste modo, será possível compreender melhor as necessidades atuais na avaliação da confiabilidade dos sistemas.

2.1 Faltas

A função de um sistema define-se como aquilo que o sistema é suposto fazer. O seu comportamento é descrito por um conjunto de estados e corresponde àquilo que este faz para implementar a sua função. Já o serviço disponibilizado por um sistema corresponde ao conjunto de funcionalidades que este proporciona aos seus utilizadores. Um serviço é disponibilizado através de uma interface e a parte do estado de um sistema (computação, informação armazenada, condição física, *etc.*) percetível na sua interface é definido como o seu estado externo [17].

A **falha** de um sistema pode ser definida como a transição do serviço correto para um serviço incorreto. Um sistema exibe serviço correto quando cumpre a sua função, definida pela sua especificação funcional, que inclui, quer as funcionalidades que o sistema deve disponibilizar, quer padrões de desempenho que deve cumprir. Isto significa que uma falha não implica necessariamente a existência de um *bug*: um atraso na entrega do serviço pode ser considerado uma falha. O desvio do serviço correto do sistema para o serviço errado é definido como **erro** e os erros são causados por **faltas**. O erro é o mecanismo através do qual uma falta se torna aparente. Uma falta encontra-se ativa quando causa um erro, caso contrário é dormente. Além disso, uma falta pode permanecer dormente durante um longo período de tempo até se manifestar como um erro. Um exemplo disso é uma função de divisão que não

verifica o valor do dividendo, já que este é um problema que passa despercebido para os casos em que o dividendo é diferente de 0. A ativação de uma falta é o processo através do qual uma falta dormente é ativada e resulta da aplicação de um determinado *input* (no caso da divisão por 0, será fornecer o 0 como dividendo) ao componente faltoso.

Como um sistema pode ser composto por um ou mais componentes, a falha de um componente não equivale, necessariamente, à falha do sistema. A falha de um sistema não ocorre desde que o estado externo do componente não faça parte do estado externo do sistema. No entanto, é possível que o erro num componente se propague para outros componentes. Isto acontece quando o erro atinge a interface do serviço do componente e outro componente utiliza esse serviço. Assim, a falha de um componente causa uma falta no sistema e esta falta poderá levar à falha do mesmo caso não seja tratada e contida adequadamente.

2.1.1 Classificação

As faltas podem ser classificadas de acordo com vários critérios: fase de origem, limites do sistema, dimensão, objetivo, intenção, persistência, causa fenomenológica e capacidade [18].

As faltas podem também ser classificadas de acordo com a reprodutibilidade da sua ativação, critério que possui especial interesse para esta dissertação. Assim, faltas cuja ativação é reprodutível são classificadas como sólidas, caso contrário, são classificadas como elusivas. As faltas elusivas são as mais comuns em sistemas de *software* complexos [17], pois as suas condições de ativação normalmente dependem de combinações complexas de estado interno e pedidos externos.

A persistência das faltas relaciona-se fortemente com a reprodutibilidade da sua ativação. Tendo como critério de classificação a sua persistência, podemos classificar as faltas como:

- Permanentes: Faltas que ocorrem uma vez e persistem até serem tratadas. No caso de faltas de *hardware*, estão relacionadas com danos em componentes de *hardware*. Por outro lado, a existência de, por exemplo, um ciclo infinito num programa é uma falta permanente de *software*. Em ambos os casos, o estado erróneo do sistema permanece até que alguém o conserte.

- Transientes: Faltas que ocorrem uma vez e desaparecem. Causadas por determinadas condições ambientais, como radiação e interferência eletromagnética, ou um *timeout* na troca de uma mensagem entre nodos devido à congestão da rede.

- Intermitentes: Faltas que ocorrem periodicamente, mas de forma irregular. Podem ser causadas por, por exemplo, degradação de componentes de conexão eletrónica. São dificilmente reproduzí-

veis e é tipicamente difícil descobrir a sua causa.

2.1.2 Confiabilidade

Um sistema confiável é um sistema capaz de disponibilizar serviços que operam segundo as expectativas do utilizador e não falham na sua normal utilização [17]. A confiabilidade é um conceito que engloba vários outros, tais como:

- **Disponibilidade**: Probabilidade do sistema estar operacional num determinado momento.
- **Fiabilidade**: Probabilidade de garantir determinados padrões de desempenho e serviços corretos durante um determinado período de tempo.

Apesar de parecidos, os conceitos de disponibilidade e fiabilidade são distintos: a disponibilidade de um sistema é medida através da prontidão (estar operacional se necessário) do sistema, enquanto que a fiabilidade é medida através da frequência e impacto das falhas [113].

As principais ameaças à confiabilidade de um sistema são as faltas, erros e falhas. Uma questão que se levanta é de que modo, perante estas ameaças, podemos desenvolver *software* confiável. Segundo Wilfredo [12] e Avizienis *et al.* [18] é possível fazê-lo através da **prevenção**, **previsão**, **tolerância** e **remoção** de faltas.

Para prevenir a introdução de faltas no desenho de *software* e *hardware*, podem ser utilizadas determinadas metodologias e técnicas, tais como modularidade, utilização de linguagens de programação fortemente tipadas, entre outras.

Já a previsão de faltas relaciona-se com a realização de estimativas do número presente, futuras incidências e consequências das faltas num dado sistema.

Apesar da existência de metodologias de prevenção de faltas, é imprudente supor que os sistemas de *software* estão desprovidos de faltas, nomeadamente em sistemas cuja complexidade é maior. Assim sendo, é necessária a implementação de mecanismos de **tolerância a faltas** que permitam minimizar ou impedir falhas no sistema.

O processo de remoção de faltas realiza-se através da verificação, diagnóstico e correção da implementação. Existem dois tipos essenciais de técnicas para verificação de *software*, que se distinguem com base na execução, ou não, de um sistema. Na verificação estática, o sistema não é executado e quando o é, a verificação diz-se dinâmica.

2.1.3 Tolerância a faltas

Um sistema é tolerante a faltas se for capaz de impedir que faltas levem a falhas. Tal é possível através da integração de mecanismos de deteção de erros e recuperação. Segundo Koren e Mani Krishna [26], a tolerância a faltas é um exercício de exploração de redundância. Esta é utilizada para compensar falhas de componentes ou tentar corrigir faltas e erros, permitindo que um sistema continue a entrega de serviço correto. Existem, essencialmente, quatro tipos de redundância:

- Redundância de *hardware*: Incorporação de *hardware* suplementar, quer para detetar, quer para mascarar os efeitos de um componente faltoso. Por exemplo, em vez de ter um processador, ter dois a executar a mesma função.

- Redundância de informação: Uma forma comummente utilizada de redundância de informação consiste na utilização de código de deteção e correção de erros. Nomeadamente, podem ser adicionados bits extra aos dados originais para que, caso um erro seja introduzido nesses dados, o erro possa ser detetado ou até corrigido. Um exemplo deste tipo de redundância que é utilizado, por exemplo, em protocolos de trocas de mensagens [71], ou até pelos SFs para consistência de dados [87], são os *checksums*.

  Outra forma comum de redundância de informação é a replicação de dados, que é frequentemente utilizada em sistemas distribuídos. Neste tipo de sistemas, os mesmos dados podem encontrar-se em diferentes nodos, o que não contribui apenas para a tolerância a faltas, mas também para o desempenho (distribuição de carga, por exemplo) [26].

- Redundância de tempo: *Timeouts* e retransmissões em comunicações fiáveis ponto-a-ponto são exemplos de redundância de tempo. Esta forma de redundância é particularmente útil na presença de faltas transientes ou intermitentes [26].

- Redundância de *software*: Uma técnica de redundância de *software* é a programação de N-versões, em que o mesmo programa é executado N vezes e a resposta da maioria é a utilizada. Outra forma deste tipo de redundância é o *software self-checking*, em que erros são detetados através de, por exemplo, asserções para verificação de resultados [37].

2.1.4 Injeção de faltas

A injeção de faltas é uma técnica de validação da confiabilidade dos mecanismos de tolerância a faltas de um sistema. Esta técnica consiste na realização de experiências onde são introduzidas faltas no sistema

alvo e na análise da sua respposta à presença destas faltas [19].

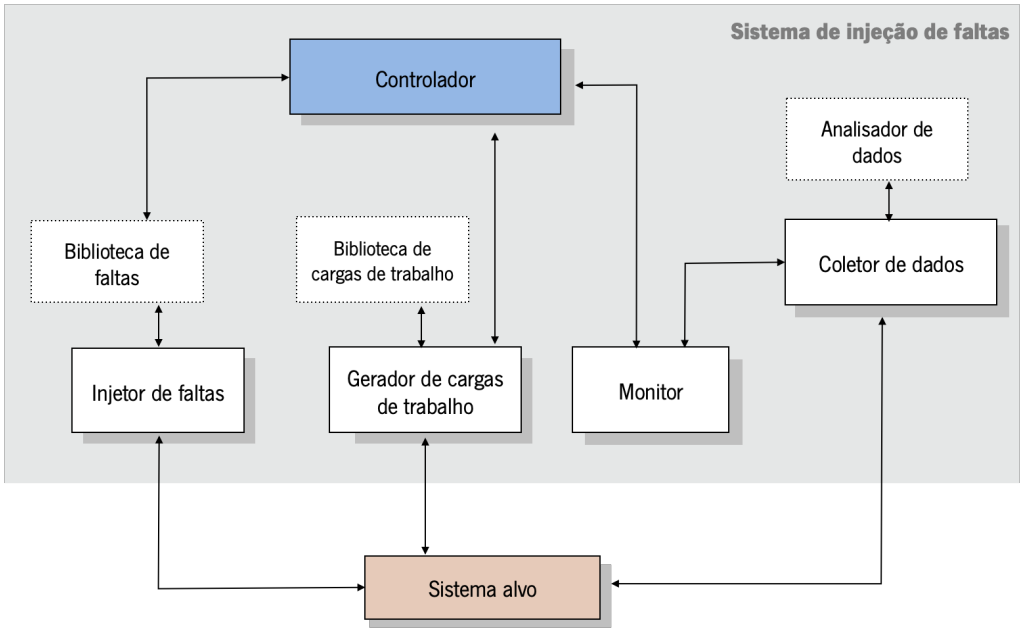


Figura 1: Arquitetura geral de um sistema de injeção de faltas.

Um ambiente de injeção de faltas tipicamente consiste em [10]:

- sistema a ser testado;
- biblioteca de faltas, que armazena diferentes faltas e semânticas de *hardware*, ou estruturas de *software*;
- gerador de cargas de trabalho, que gera cargas de trabalho para o sistema sob teste;
- biblioteca de cargas de trabalho, responsável por armazenar as diferentes cargas de trabalho;
- controlador, que controla a experiência;
- injetor de faltas, que tem o papel de injetar faltas à medida que invoca comandos do gerador de cargas de trabalho;
- monitor, responsável por rastrear a execução dos cargas de trabalho e iniciar a coleção de dados;
- coletor de dados;
- analisador de dados, que processa e analisa os dados.

As faltas injetadas podem ser de diferentes tipos e podem ser injetadas em locais e momentos diferentes. A altura em que um falta é injetada, a sua duração, o tipo de falta e o local dentro do sistema onde é injetada definem o modelo de faltas [13, 52].

Classificação

As duas principais técnicas de injeção de faltas são a injeção baseada em *hardware* e injeção baseada em *software*. A escolha do tipo de técnica mais adequada depende de vários fatores como o custo, o nível de perturbação, a acessibilidade a pontos de injeção, a controlabilidade, a reprodutibilidade e a portabilidade.

A injeção baseada em *hardware* [1, 2, 3, 36] é realizada através da introdução de distúrbios ao nível do *hardware*. Duas das desvantagens deste tipo de injeção é o facto de ser necessário *hardware* adicional, que é responsável por desencadear as faltas e monitorizar o seu impacto, e a possibilidade de danificar o *hardware* com as experiências.

A injeção baseada em *software* [4, 8, 9, 14] não necessita de *hardware* adicional, o que é vantajoso a nível de custos, e pode ser direcionada às aplicações e SOs alvo, o que é difícil de fazer com a injeção de faltas baseada em *hardware*. Para além disso, esta técnica ultrapassa problemas relacionados com a controlabilidade e repetibilidade das experiências, que existiam com a técnica de injeção baseada em *hardware*.

Vários tipos de faltas podem ser injetadas através desta técnica, desde faltas de registos e de memória, à perda ou duplicação de pacotes de rede. É de realçar que é possível injetar faltas ao nível dos dados de um programa (através da modificação dos dados na *heap*, na *stack* e nos registos), no código do programa e nas suas interfaces (*e.g.* chamadas ao sistema, *Interprocess Communication* (IPC), *Remote Procedure Call* (RPC)) [53]. Isto significa que existe a possibilidade de efetivamente causar faltas no *software*, fazendo alterações ao código, mas também de emular apenas os efeitos das faltas de *software*, como forçar o lançamento de um erro, sem que tenham existido efetivamente condições para este ser lançado.

Existem duas abordagens dentro da injeção de faltas baseadas em *software*: injeção em **tempo de compilação** ou em **tempo de execução**.

Na injeção de faltas em tempo de compilação, as faltas são injetadas através da modificação do código fonte ou do código *assembly* de um programa, sendo, portanto, uma técnica com baixa portabilidade. A falta é ativada quando o sistema é executado, não existindo nenhum mecanismo específico necessário para ativação de faltas, ao contrário da injeção de faltas em tempo de execução, em que tipicamente são

utilizados os seguintes mecanismos de ativação de faltas:

- ***Timeout***: A expiração de um temporizador de *software* ou *hardware* desencadeia a injeção de uma falta.
- **Exceção/*trap***: As faltas são injetadas segundo a ocorrência de determinados eventos, como exceções de *hardware* ou *software traps*. No caso das exceções de *hardware*, a injeção ocorre devido à observação de um evento de *hardware*, tal como o acesso a um dado endereço de memória. Já no segundo caso, a inserção de uma instrução de *software trap* num programa, por exemplo, resultará na invocação do injetor quando o programa executar uma dada instrução. A injeção de faltas segundo esta técnica é mais previsível do que a anterior, pois não existem dependências temporais.
- **Inserção de código**: Adição de instruções ao programa a ser testado (as instruções originais não são modificadas).

2.1.5 Reprodutibilidade

Um dos principais problemas com as estratégias de injeção de faltas baseadas em *hardware* é a dificuldade de reprodução dos testes. Mesmo em estratégias baseadas em *software*, é comum utilizar-se a aleatoriedade, um inimigo da reprodutibilidade, na tentativa de encontrar o maior número possível de vulnerabilidades nos sistemas. Algumas das ferramentas de injeção de faltas em sistemas de armazenamento mais conhecidas, como o Jepsen [100] e o Chaos Monkey [85], injetam faltas e geram cargas de trabalho de forma aleatória.

A reprodutibilidade é um dos pilares das pesquisas científicas, contribuindo para a garantia de que os resultados são confiáveis e a compreensão da causa de um determinado fenómeno. Segundo Vitek e Kalibera [31], muitos estudos recentes carecem deste tipo de características: repetibilidade e reprodutibilidade. Já Hunt *et al.* [35] mencionam que sem a capacidade de reproduzir de forma precisa uma execução que possui *bugs*, a tarefa de descobrir a causa do problema é dificultada. Um teste de injeção de faltas ser reprodutível significa, por exemplo, uma falta ser introduzida sempre que uma dada condição é verificada e sempre no mesmo estado do sistema alvo.

A instrumentação de sistemas através de testes aleatórios nem sempre é a mais adequada. Por vezes, é necessário um nível superior de controlo para forçar a execução segundo um determinado caminho e repetir esse caminho várias vezes para a falta ser devidamente identificada e corrigida. Para isso, é necessário conhecer bem a carga de trabalho utilizada, a falta injetada, o ponto de injeção e a reação do

sistema a essa falta. Existem outras ferramentas de âmbito exploratório que não recorrem à aleatoriedade, mas sim a estratégias bem definidas de exploração de estados de falha dos sistemas, como o ALICE [39]. No entanto, no caso desta ferramenta, não existe, aparentemente, uma forma simplificada de reproduzir um determinado problema.

É inegável que ferramentas de exploração ajudam na construção de confiança sobre os sistemas testados, pois permitem encontrar problemas que passaram despercebidos. Mas a realidade é que muitas aplicações dependem dos problemas reportadas pelos utilizadores para a sua correção. É comum os *releases* corrigirem uma grande parte de problemas que só os utilizadores foram capazes de detetar. Quando um utilizador expõe um problema que detetou num dado sistema, seria de interesse ser possível reproduzir esse problema em particular e, eventualmente, reproduzi-lo de forma automática em cadeias de teste.

2.2 Sistemas de armazenamento

Esta dissertação foca-se na injeção de faltas baseada em *software* em sistemas de armazenamento locais. Apenas serão consideradas faltas que causam perda de dados em memória volátil, ou seja, faltas como falhas de energia. É de realçar que existem vários tipos de faltas que podem ser injetadas ao nível da pilha de armazenamento, tais como corrupção de dados, erros de Entrada/Saída (E/S), atrasos de operações de leitura ou escrita, entre outros, que são ortogonais a este estudo [55, 83].

Nesta secção será explorado o fluxo dos dados num SO tradicional. Este estudo é essencial dado que, para compreender o impacto da perda dos dados nas aplicações, é necessário perceber quais dados podem ser perdidos e exatamente em que camadas da pilha de armazenamento tal pode acontecer.

Dispositivos de armazenamento

Quando uma falha de energia ocorre, as aplicações apenas podem contar com os dados que foram guardados nos dispositivos que permitem armazenar dados de forma persistente. Atualmente, existem vários tipos destes dispositivos [104], mas nesta dissertação apenas iremos considerar os tradicionais dispositivos de armazenamento como os HDDs e SSDs. Tanto os HDDs como os SSDs fazem parte de uma classe de dispositivos de armazenamento chamados *block devices*, em que os dados são transferidos em blocos. Um bloco é um segmento contíguo de *bytes* de tamanho fixo. O bloco possui um tamanho físico, que corresponde ao mínimo que pode ler lido ou escrito para um dispositivo de armazenamento. O tamanho de um bloco varia conforme o tipo e o modelo do dispositivo de armazenamento. Os HDDs tradicionais, por exemplo, possuem blocos (também denominados setores) de 512 *bytes*. Isto significa

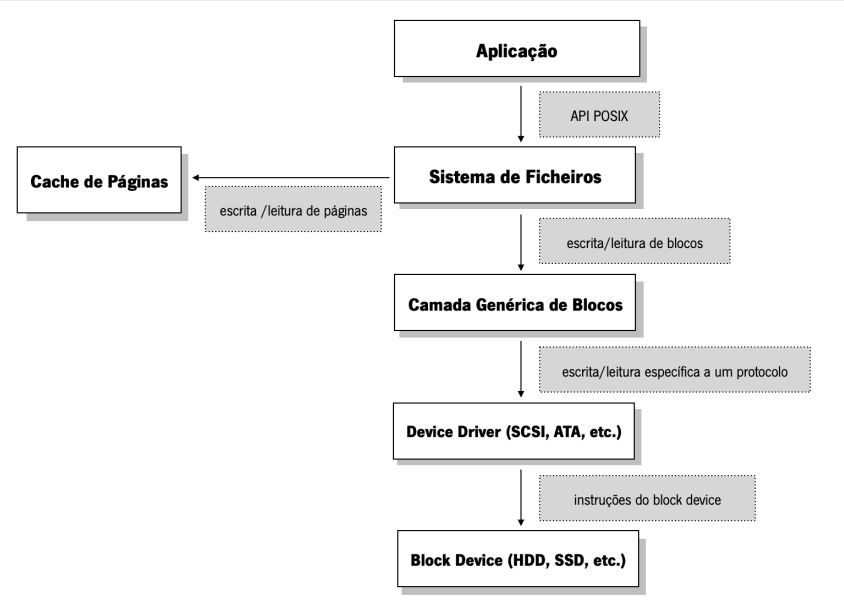


Figura 2: Pilha de armazenamento tradicional.

que 512 *bytes* são escritos **atomicamente**, isto é, ou a totalidade dos 512 *bytes* será escrita, ou nenhum *byte* será escrito, não existindo estados intermédios.

Sistemas de ficheiros

Os SFs representam uma abstração para o *hardware* responsável pelo armazenamento de blocos de dados. Usualmente, o tamanho de um bloco de dados de um SF é um múltiplo do tamanho do bloco físico (o usual é 4096 *bytes*). Num SF existem duas entidades principais: os ficheiros e as diretorias. O ficheiro é um *array* linear de *bytes*, que pode ser lido e escrito. Para além dos dados, um ficheiro tem também associado a si metadados, tais como o nome, identificador (número do *inode*), tipo, tamanho, as permissões de acesso, entre outros. A estrutura de dados do SF que guarda os metadados de um ficheiro é denominada *inode*. Por sua vez, uma diretoria é uma entidade que "contém"ficheiros, isto é, é uma lista de pares com os nomes dos ficheiros e os seus identificadores únicos.

Os SFs estão usualmente organizados nas seguintes estruturas de dados em disco: *inodes*, blocos de dados, estrutura de alocação (*bitmap*, por exemplo) e superbloco.

As estruturas de alocação permitem gerir quais *inodes* e blocos de dados estão alocados ou livres. Já o superbloco mantém informações gerais sobre o SF em questão, tal como quantos *inodes* e blocos de dados existem no sistema.

2.2.1 Fluxo de dados

Os SFs disponibilizam uma interface para a manipulação de ficheiros, que inclui métodos de atualização de dados e metadados. Nesta dissertação iremos apenas considerar interfaces compatíveis com POSIX.

As operações de leitura e escrita de ficheiros solicitadas pelas aplicações são traduzidas em operações sobre blocos pelo SF. Em sistemas Linux, estas operações são reencaminhadas, por uma camada genérica de blocos, para o respetivo *device driver*, visto que o SF é alheio às especificações dos dispositivos. São os *device drivers* os responsáveis por efetivamente transferir os dados de e para os dispositivos periféricos (como um HDD). Estas operações de transferência de dados entre o computador e dispositivos de E/S (dispositivos que permitem leitura e escrita de dados) são tipicamente denominadas de operações de E/S [15].

As operações de leitura e escrita de/para um ficheiro são custosas devido aos acessos a disco serem lentos [24]. Para além disso, várias operações de E/S são executadas no seu fluxo de execução. Uma escrita para um ficheiro não envolve apenas a escrita em disco dos blocos de dados, mas também a atualização das estruturas necessárias para a gestão dos ficheiros. Assim, uma escrita divide-se em três operações de E/S: uma para ler a estrutura de alocação, outra para atualizá-la e outra para escrever efetivamente os blocos de dados [58].

De modo a colmatar este problema de desempenho, a maior parte dos sistemas utiliza a memória volátil para guardar dados, já que o tempo de acesso à *Random Access Memory* (RAM) é muito menor. Este processo de guardar em memória dados que potencialmente podem ser acedidos novamente num curto espaço de tempo é denominado *caching*. O *kernel* do Linux possui uma cache de disco denominada Cache de Páginas [23].

As páginas da cache dos sistemas Linux são blocos de dados de tamanho fixo, normalmente de 4096 *bytes*, e podem ser de diferentes tipos, tendo em conta o tipo de dados que possuem, dado que podem possuir dados de ficheiros regulares, de diretorias, ou dados diretamente lidos de ficheiros dos *block devices*, entre outros [22].

Quando, por exemplo, uma aplicação solicita uma escrita para um ficheiro, primeiro estes dados são guardados em páginas na cache e, só posteriormente, é que passam pelas restantes camadas da pilha de armazenamento. Estas páginas são marcadas como sujas de modo a identificar os dados nelas presentes como ainda não persistidos em disco. As escritas realizadas para a cache de páginas são chamadas de assíncronas, já que as aplicações recebem a confirmação de execução de uma escrita quando esta é realizada para a memória e, só posteriormente, será efetivamente persistida.

Quando são realizadas leituras, são também criadas páginas com dados e que, ao contrário das

páginas criadas pelas escritas, não são marcadas como sujas, já que os dados foram lidos diretamente do disco. Assim, por exemplo, se forem solicitadas duas leituras consecutivas da mesma porção de um ficheiro, na segunda leitura, os respetivos blocos de dados encontrar-se-ão em *cache*, não existindo necessidade de realizar um acesso a disco.

Em geral, existem vários benefícios, para além de evitar acessos a disco, dos quais os SOs tiram partido através do *caching* de escritas [58]:

- Os SFs podem agrupar as atualizações em conjuntos mais pequenos de forma a realizar menos pedidos de E/S: se, por exemplo, é realizada uma escrita para um ficheiro seguida de outra, ao agrupar ambas basta atualizar apenas uma vez o *inode* do respetivo ficheiro.
- Os pedidos de E/S podem ser agendados por um *disk scheduler* do SO de modo a aumentar o desempenho. Os pedidos são analisados e ordenados utilizando diferentes políticas (*Shortest Seek Time First*, SCAN, *etc.*).
- Algumas operações de E/S podem ser completamente evitadas, como no caso de um ficheiro ser criado e logo depois eliminado.

Existem também camadas abaixo do SF, como a camada de blocos, os *device drivers* e os controladores de disco, que fazem as suas próprias otimizações através do *caching*, *batching* e reordenação de escritas de blocos de dados [46].

2.2.2 Persistência de dados

De modo a impedir que os dados fiquem indefinidamente em memória volátil, os SOs possuem mecanismos para fazer *flush* das páginas sujas da cache de páginas para disco. No caso dos sistemas Linux, por exemplo, existe um conjunto de *kernel threads* (`pdflush`) responsáveis por sincronizar periodicamente, normalmente a cada 30 segundos, os dados em memória persistente com os dados em memória volátil [94]. Apenas escrevendo dados em disco é que se previne a sua perda na sequência de uma falha de energia ou de falha do sistema.

Algumas aplicações, tais como bases de dados, possuem políticas restritas relativas a ordenação e persistência de dados. De modo a evitar perdas, as escritas necessitam de ser forçadas para disco. Uma opção para forçar essa sincronização é a utilização da interface do *raw disk*, que permite o acesso direto a um dispositivo secundário de armazenamento, evitando por completo o SF e, consequentemente, o *caching*. Esta era uma técnica antigamente utilizada por algumas bases de dados, quer por razões de

fiabilidade, quer por eficiência e desempenho. No entanto, com a evolução dos SF e o surgimento do E/S direto, deixou de ser tão utilizado [80]. O E/S direto permite contornar algumas funcionalidades do SF, como a cache de páginas, sem a necessidade de o evitar completamente, ou seja, é possível operar sobre ficheiros, ao contrário da opção do *raw disk*, em que se opera diretamente sobre blocos.

Para além de os SFs disponibilizarem *flags* na chamada ao sistema `open()` para permitir que as escritas num ficheiros sejam síncronas [103], existem também algumas chamadas ao sistema POSIX que permitem que um processo solicite o *flushing* dos blocos sujos do *kernel* do SO para disco:

- **`sync()`**: *flush* dos blocos sujos de todos os ficheiros.
- **`fsync()`**: *flush* de todos os blocos sujos (de dados e metadados) de um dado ficheiro.
- **`fdatasync()`**: muito parecida ao `fsync()`, mas não faz *flush* da informação relativa ao *inode*, ou seja, apenas os dados (e não os metadados) são *flushed*.

São estas as funções normalmente utilizadas pelas aplicações para forçar as escritas críticas. São uma boa alternativa a *flags* como `O_SYNC` (em que todas as escritas de um dado ficheiro são síncronas), visto que, para algumas aplicações, não é fundamental para a sua resiliência que todas as escritas sejam persistidas de imediato.

2.2.3 *Crash consistency*

Um sistema ou uma aplicação é *crash-consistent* se é capaz de recuperar para um estado válido após a ocorrência de uma falha de energia ou do SO (situações em que os dados em memória volátil são perdidos), sem que os dados essenciais tenham sido perdidos ou corrompidos [42].

O problema de *crash consistency* advém, essencialmente, da utilização da cache, pois as escritas são dadas como concluídas quando são realizadas para a cache. Assim, na perspetiva do utilizador, não existem estados intermédios, apesar de uma escrita poder estar apenas parcialmente persistida em disco. No entanto, estes estados intermédios ficam visíveis quando ocorrem falhas.

Ao nível do SF e cache de páginas, estas atualizações parciais do disco podem ocorrer quando:

- A cache de páginas fica sem espaço. Quando tal acontece, são aplicadas diferentes políticas de substituição da cache, que podem levar a que algumas páginas sujas de um ficheiro tenham sido escritas depois de outras, mas por serem menos usadas, por exemplo, sejam persistidas antes.
- Os dados em cache estão a ser *flushed*. A ordem em que as páginas são persistidas pode ser diferente da ordem em que foram escritas.

A utilização da cache surge de um compromisso entre desempenho e fiabilidade: as escritas para disco são adiadas, de modo a que o desempenho seja melhorado, ao custo da possibilidade de os dados serem perdidos na eventualidade de uma falha de energia ou falha do sistema.

Sistemas de ficheiros

Os SFs, que estão encarregues da gestão dos dados guardados nos dispositivos de armazenamento, possuem estruturas de dados cruciais, cuja integridade deve ser garantida na eventualidade de uma falha de energia. É possível que o SF fique corrompido, caso estas estruturas fiquem apenas parcialmente atualizadas. Por exemplo, pode acontecer que apenas o *inode* seja escrito para disco. Nesta situação, o *inode* aponta para um bloco em disco que contém lixo (*i.e.*, dados antigos) e a estrutura de alocação contém informação de que o bloco não foi alocado.

Os primeiros SFs não possuíam meios de prevenir a ocorrência de problemas de *crash consistency*, apenas formas generalizadas de reparar os estragos. Assim, após a ocorrência de uma falha de energia, ou de qualquer outro problema com o SF, era utilizada a ferramenta fsck [120] (para o caso de sistemas UNIX, mas para outros sistemas existem ferramentas semelhantes), que deteta e repara inconsistências. Um dos principais problemas com esta ferramenta é o tempo necessário para analisar o disco, o que, com o aumento do tamanho do armazenamento, se tornou cada vez mais impraticável.

Atualmente, já existem técnicas [7, 16] e SFs mais sofisticados [34, 75] que permitem a prevenção destas inconsistências. Um exemplo dessas técnicas, que é disponibilizada por SFs comummente utilizados (ext3, ext4, NFTS, APFS, etc.), é o *journaling* [58]. Com *journaling*, em vez de as estruturas de dados serem diretamente alteradas, estas alterações são primeiramente escritas num ficheiro à parte (o *journal* ou log) como uma transação, isto é, um conjunto de operações que é aplicado na totalidade, nunca parcialmente. Só depois de uma transação estar devidamente persistida em disco, o que também é, em si, um desafio, é que as alterações são aplicadas às estruturas de dados do SF. Se uma falha de energia ocorrer durante a atualização das estruturas do SF, basta refazer ou fazer *rollback* da transação escrita no log.

É também possível que ocorra uma falha de energia durante a atualização do próprio log, podendo deixá-lo num estado inconsistente. As *write barriers* podem ser utilizadas para garantir a ordem e durabilidade das escritas destas transações no log [27, 86]. Para além disso, uma forma de detetar inconsistências no log passa pela utilização de *checksums* nas transações no log [20]. Assim, no processo de recuperação do SF, se houver uma incompatibilidade entre o *checksum* calculado e o presente, esta transação é descartada.

Aplicações

Também as aplicações estão suscetíveis a estados inconsistentes após a ocorrência de falhas de energia ou do SO. A maioria das aplicações não especifica o tipo de SF ou o tipo de configurações que este deve ter para que a aplicação execute sem que ocorram problemas de consistência. Existem várias possíveis razões para isso.

Por um lado, é desejável que as aplicações sejam portáveis, de modo a que possam ser utilizadas sobre diferentes SFs. Por defeito, a maior parte dos SFs apenas realiza *journaling* de metadados, o que pode levar à corrupção dos ficheiros de dados da aplicação.

Por outro lado, mesmo com *journaling* de dados, os SFs não garantem a *crash consistency* da aplicação, pois a aplicação pode possuir as suas próprias transações. Para aplicar estas transações, as aplicações necessitam de propriedades específicas dos SFs. No entanto, as garantias dos SFs relativamente a ordenação e atomicidade das escritas são subespecificadas e, frequentemente, não intuitivas [39, 42, 46, 67].

Existem vários artigos científicos a expor o problema de *crash consistency* das aplicações e a apresentar soluções cada vez mais eficientes, que passam até por desenvolver novos SFs, para este problema [57, 62, 69, 73]. Uma das estratégias mais comuns adotadas pelas aplicações para tolerar inconsistências é o *Write-Ahead Logging* (WAL), que é semelhante ao *journaling*. É comum as aplicações escreverem as alterações para o log e, após algum tempo, executarem um *checkpoint* [108] onde as transações do log são realmente aplicadas nos ficheiros da aplicação.

As aplicações tipicamente possuem outros ficheiros para além daqueles que contêm dados e, normalmente, o WAL é utilizado apenas para dados. Para garantir que não ocorram escritas parciais, é típico utilizar-se uma sequência típica de chamadas ao sistema POSIX [42]: `write` para um ficheiro temporário, `fsync`, `rename` do ficheiro temporário para o nome do ficheiro final e `fsync` da diretoria. Dado que o `rename` é atómico [114], o ficheiro final terá todo o conteúdo, ou nenhum.

É de realçar, no entanto, que em alguns sistemas operativos, a execução de comandos de sincronização, como o `fsync`, pode não ser suficiente para evitar problemas de *crash consistency* [27]. Isto acontece porque o `fsync` é dado como concluído quando os dados são transferidos para a cache volátil do disco.

Apesar do problema de *crash consistency* ser bem conhecido, há inúmeros *bugs* reportados, quer em sistems de ficheiros [56], quer em aplicações [39, 92, 119]. Construir sistemas *crah-consistent* é uma tarefa difícil, pois, devido à complexidade crescente das aplicações, os protocolos de atualização dos dados tornam-se, também eles, mais complexos e minuciosos. Consequentemente, o trabalho de garantir

a consistência desses dados é também mais complicado. Pequenos erros podem levar à corrupção ou perda de dados silenciosa, o que torna indispensável testar a *crash consistency* dos sistemas [56].

Os problemas de *crash consistency* das aplicações não são fáceis de reproduzir, pois é necessário imitar comportamentos muito específicos do SF. Assim, surge a necessidade de desenvolver ferramentas que possibilitem a simulação destes comportamentos e a reprodução de problemas que advêm destes.

2.3 Trabalho relacionado

Até ao momento, foram desenvolvidas várias ferramentas para testar a *crash consistency* das aplicações e SFs. Pretende-se, com este capítulo, mencionar algumas dessas ferramentas, especificando os seus objetivo e funcionamento. Deste modo, será possível perceber os aspetos que a solução desenvolvida pode abordar e que não foram totalmente cobertos por outras ferramentas.

2.3.1 *Crash consistency* em sistemas de ficheiros

Um conjunto de trabalhos foca-se em perceber quais as vulnerabilidades dos SFs face à ocorrência de falhas de energia, com o propósito de oferecer uma visão mais clara do que as aplicações e os utilizadores podem esperar.

O trabalho realizado por Martinez e Chidambaram [56] consiste em testar SFs de forma não intrusiva, utilizando a técnica de testes caixa negra. Esta técnica consiste em perturbar e observar o sistema alvo sem conhecimento do seu estado interno, focando-se no *output* do sistema, em vez dos seus atributos internos. Daqui resultaram duas ferramentas distintas, o **ACE** para gerar cargas de trabalho e o **CrashMonkey** para injetar as faltas no sistema alvo e verificar o seu estado após as faltas.

O CrashMonkey recebe as cargas de trabalho geradas pelo ACE e gera um conjunto de estados de falha. Depois, um *checker* compara os ficheiros persistidos em cada estado com o esperado e reporta um problema caso haja uma incompatibilidade. As cargas de trabalho são testadas pelo CrashMonkey através da simulação de uma falha após pontos de persistência, como a chamada ao sistema `fsync`. Para simular uma falha, é utilizada uma técnica denominada *record-and-replay*. Esta técnica, tal como o próprio nome indica, consiste em duas fases distintas: na primeira é aplicada uma carga de trabalho ao sistema e são recolhidas informações, como operações do SF e pedidos de E/S, e na segunda os pedidos de E/S são repetidos até um determinado ponto de persistência. O CrashMonkey depois monta o SF no estado de falha gerado, permitindo que o SF recupere. De modo a comparar o estado recuperado do SF com um estado correto, o CrashMonkey também captura uma imagem do SF após o ponto de persistência,

realizando um `unmount` seguro para permitir que quaisquer operações pendentes sejam executadas.

O trabalho realizado por Bornholt *et al.* [46] consiste na definição de modelos de *crash consistency*, isto é, modelos que descrevem o estado dos SF após a ocorrência de uma falha. É realizada uma especificação de modelos de *crash consistency* dos SF através de:

- Testes *litmus*: Pequenos programas que demonstram os comportamentos permitidos e não permitidos dos SF na sequência de uma falha.
- Especificações formais: Descrições axiomáticas e operacionais do comportamento de *crash consistency*.

O **Ferrite** é um *toolkit* para testar os modelos de *crash consistency* desenvolvidos contra SFs reais, recorrendo à execução de testes *litmus*. Este explora todos os comportamentos possíveis de falha de um dado teste *litmus*, através de enumeração (execução de testes em implementações de SFs) e verificação de modelos (execução de testes em especificações formais). Os estados de falha abordam o problema de atomicidade e reordenação, pois transformam um conjunto de chamadas ao sistema em operações atómicas (um `write` em escritas para vários blocos) e permitem a sua reordenação.

Um dos objetivos deste trabalho é mostrar a utilidade dos modelos de *crash consistency* para os programadores de aplicações. Para isso, foram desenvolvidas duas aplicações distintas: um verificador que prova se um programa fornece determinadas garantias relativamente a um dado modelo de *crash consistency* e um sintetizador que insere o mínimo de invocações da chamada `fsync` para que o programa satisfaça determinadas garantias respetivamente a um modelo. No estudo realizado não existem evidências destas ferramentas terem sido aplicadas em sistemas de armazenamento reais.

2.3.2 Exploração de estados de falha em aplicações

Uma grande parte das ferramentas para testar sistemas de armazenamento tem como objetivo automatizar o processo de encontrar vulnerabilidades. Deste modo, implementam estratégias para gerar e explorar diversos estados de falha. Tanto o Ferrite, como o CrashMonkey e ACE, mencionados acima, o fazem, mas ao nível do SF.

A ferramenta desenvolvida por Zheng *et al.* [40] permite verificar as propriedades Atomicidade, Consistência, Isolamento e Durabilidade (ACID) de bases de dados. As falhas de energia são modeladas como a perda de quaisquer operações ao nível dos blocos que não foram confirmadas como persistidas em armazenamento persistente. Se uma operação é perdida, todas aquelas que dependam explicitamente

dela também serão perdidas. Os pontos de injeção são escolhidos segundo duas políticas: uma é exaustiva e a outra baseia-se em padrões observados em duas bases de dados testadas que levaram à violação de propriedades ACID.

A ferramenta é constituída por três componentes principais: o *worker*, o *checker* e *record-and-replayer*. O *worker* exercita a base de dados, sobre uma imagem de disco bem conhecida, com cargas de trabalho que foram desenhadas especificamente com o propósito de exercitar as garantias ACID. O *record* coleta os *traces* de operações sobre blocos intersetando a camada iSCSI. O *replayer* depois simula uma falha de energia reproduzindo parcialmente as operações sobre blocos, criando uma nova imagem de disco. Posteriormente, o *checker* acede à base de dados, sobre a nova imagem de disco produzida, e as propriedades ACID são verificadas.

Pillai *et al.* [39] realizou um estudo das propriedades de persistência de diferentes SFs e desenvolveu uma ferramenta, o **ALICE** (*Application-Level Intelligent Crash Explorer*), para analisar os protocolos de atualização das aplicações e encontrar determinados requisitos de persistência que estas necessitam para funcionar corretamente, apesar de os SFs tipicamente não os garantirem.

O ALICE constrói os diferentes estados em que um ficheiro se pode encontrar em disco depois de uma falha e testa a correção da aplicação perante cada um desses estados. Os estados de falha dependem do SF, por isso o ALICE usa *Abstract Persistence Models* (AMPs), que definem os estados de falha possíveis num dado SF. As chamadas ao sistema das aplicações são transformadas em micro-operações que, por sua vez, são "anotadas" com restrições do AMP, que definem quais micro-operações podem chegar ao disco antes de outras. Para simular os estados de falha e testar as aplicações, o ALICE utiliza a estratégia de *record-and-replay*, já mencionada anteriormente.

Com o objetivo de verificar se os protocolos de atualização de dados funcionam corretamente na presença de *correlated crashes*, foi desenvolvida a ferramenta **PACE** (Protocol-Aware Crash Explorer) [44]. Estas falhas são caracterizadas pela ocorrência simultânea da falha de todas as réplicas de um *data shard* em particular e podem ocorrer, por exemplo, devido a uma falha de energia a nível de um centro de dados inteiro. O PACE pretende analisar de que forma os protocolos globais de recuperação interagem como os protocolos locais de armazenamento na presença destas faltas. Estes protocolos locais dependem do modo como as operações são persistidas pelos SFs.

O PACE explora os vários estados persistentes possíveis após a ocorrência de uma *correlated crash*. O PACE baseia-se no ALICE, sendo o seu funcionamento muito semelhante. No entanto, o ALICE tem como alvo de estudo vulnerabilidades de falha num sistema de armazenamento local, não tendo em consideração dependências entre nodos, que são características dos sistemas distribuídos.

Ainda no âmbito dos sistemas distribuídos, Gunawi *et al.* [30] conduziram um estudo para testar o funcionamento dos sistemas de nuvem face a diferentes cenários de falha: falha do nodo, falha permanente de disco, corrupção de disco, partição de rede ao nível do nodo ou *rack* e falha transiente. Neste sentido, foi desenvolvido o **FATE** (*Failure Testing Service*) que exercita de forma sistemática os sistemas de nuvem através da sua submissão a diversos cenários de falha. Um dos seus principais objetivos é testar quantas combinações de falhas forem possíveis, mas, em oposição a abordagens aleatórias, é utilizada uma técnica de redução do número de cenários a ser testados, de modo a que a testagem de diferentes cenários seja priorizada. Para a implementação desta técnica são utilizados identificadores para os estados explorados, denominados por IDs de falha. O FATE, por possuir estes mecanismos de identificação, permite a injeção de apenas um conjunto de faltas como, por exemplo, falhas em pontos de E/S específicos, apenas faltas de *crash* do nodo, *etc.* As faltas são injetadas através duma camada, entre a biblioteca do SO e o sistema sob teste. Esta camada é responsável por gerar os ID de E/S e verificar se uma falha permanente já afetou esse ponto de E/S.

Para validar o comportamentos dos sistemas de nuvem no processo de recuperação, é especificado o comportamento expectável através do **DESTINI** (*Declarative Testing Specifications*), que se baseia na linguagem Datalog. Assim, por um lado, temos o FATE que causa falhas no sistema e, por outro, o DESTINI, que permite analisar se os protocolos de recuperação atuam como o esperado.

O **EXPLODE** [25] é uma ferramenta que tem como objetivo verificar sistemas de armazenamento como SFs, aplicações, máquinas virtuais, *etc.*, na sequência de falhas.

Os utilizadores devem fornecer três componentes específicos ao sistema a ser testado. O primeiro componente é um *checker*, que realiza operações do sistema de armazenamento, e onde o utilizador deve especificar pontos em que existem diferentes caminhos de execução, para que o EXPLODE os gere e teste. O segundo componente serve para configurar o sistema, e o último permite combinar diferentes *checkers* num único, permitindo testar uma pilha inteira de armazenamento.

Os estados de falha são gerados através da construção de um conjunto de blocos de dados e da escrita para disco de qualquer subconjunto deste. Para além das falhas em que a memória volátil é perdida, o EXPLODE também estuda o impacto de falhas menos graves, como a situação em que a aplicação falha, mas o SO não.

O trabalho desenvolvido por Koskinen e Yang [50] especifica a recuperação de falhas. Após uma falha, o programa deve alcançar um estado equivalente a um estado que foi visitado antes da ocorrência dessa falha. O principal foco do trabalho é apresentar e aplicar uma técnica que permite, de forma automática, apresentar provas de recuperabilidade. O método utilizado analisa programas que podem

ser traduzidos em autómatos de controlo de fluxo e reduz a recuperabilidade de uma falha ao problema de alcançabilidade, ou seja, se o fluxo de controlo não alcança um estado de erro, é correto. Para tal, foi desenvolvida a ferramenta **ELEVEN82**, que prova a recuperabilidade de programas, escritos em C e C++, que implementam protocolos complexos de atualização de dados como WAL. O facto de os sistemas testados necessitarem de ser escritos numa dada linguagem constitui uma grande limitação desta ferramenta.

O ALICE também inspirou a criação da ferramenta **C**^{3} [49]. As aplicações estudadas por esta ferramenta são menos complexas do que sistemas de armazenamento (*e.g.*, editores de texto). O método utilizado para gerar estados de falha é semelhante ao ALICE, no entanto, contrariamente a este, foca-se na necessidade de reduzir o trabalho do utilizador. Por um lado, evita escrita de *checkers* para os sistemas e, por outro, faz uma amplificação dos testes.

2.3.3 Injetores de faltas

O **CORDS** [55] possibilitou a descoberta de vários problemas com os mecanismos de tolerância a faltas de SFs em sistemas de armazenamento distribuídos. Foi constatado que a ocorrência de uma falta num único nodo do sistema poderia causar problemas como perda de dados, corrupção e indisponibilidade a nível global.

O CORDS é constituído por dois componentes: o *errfs*, um sistema de ficheiros *File System in User Space* (FUSE), responsável por injetar faltas, e o *errbench*, que corresponde a um conjunto de cargas de trabalho e *scripts* de inferência de comportamento para cada sistema testado.

As faltas que podem ser injetadas com o CORDS são a corrupção ou erro (de E/S ou de espaço) de blocos do SF e a corrupção de bits. Estas faltas são injetadas em estruturas de dados ao nível da aplicação e não em estruturas de dados do SF, como os metadados. Para as corrupções, o *errfs* altera o conteúdo de um bloco antes de o retornar para a aplicação. Para simular erros, o *errfs* apenas retorna o respetivo código de erro, ou seja, não simula exatamente o erro.

Em cada momento, apenas uma falta é injetada, em apenas um bloco e apenas num nodo, de modo a dar mais margem de recuperação para as aplicações. Mesmo com esta simplificação, foi demonstrado que várias aplicações falham na tarefa de recuperação e, em alguns casos, o erro é propagado a outras instâncias do conjunto de nodos.

Existem, atualmente, alguns injetores de faltas [48, 63, 84, 101, 105] que, tal como o CORDS, se baseiam em SFs em espaço de utilizador, já que é possível intersetar todas as operações importantes relacionadas com a manipulação de dados de ficheiros, sendo possível, por exemplo, corromper alguns

desses dados, atrasar escritas e leituras, *etc.*

Um exemplo desses injetores de faltas é o **UnreliableFS** [84], um sistema de ficheiros FUSE de simples configuração, que permite a injeção das seguintes faltas:

- retorno de valores de erro;
- envio de `SIGKILL` ao processo que invoca a operação;
- substituição de uma operação por outra que não faz nada;
- atraso na execução da operação invocada.

Para cada tipo de falta é especificada uma probabilidade para a sua ocorrência e para o caso da falta de atraso de uma operação, a duração desse atraso deve ser especificado.

Outros dois sistemas de ficheiros FUSE para injeção de faltas são o **CharydbeFS** [48] e o **PetardFS** [105]. Ambos simulam a ocorrência de erros. No caso do PetardFS, um ficheiro XML é fornecido com as configurações necessárias, incluindo as faltas a injetar. É possível definir o ficheiro e os respetivos *bytes* em que ocorrerá um determinado erro. Já o CharydbeFS foi desenvolvido para testar a base de dados distribuída ScyllaDB [116] e, contrariamente ao PetardFS, permite a progrmação de faltas enquanto a aplicação está a ser executada. É possível, por exemplo, terminar a execução de uma base de dados quando uma dada operação é executada, injetar erros segundo determinadas condições, atrasar operações e definir probabilidades para a ocorrência de qualquer uma destas faltas.

O estudo desenvolvido por Rebello *et al.* [63] tem como objetivo estudar o comportamento dos SFs e das aplicações na sequência das falhas de `fsync`, nomeadamente o erro de `EIO`, que ocorre após o SF começar a sincronizar as páginas sujas para o disco. Este erro é mais difícil de lidar, porque algum estado já pode ter sido alterado e pode ou não ser possível revertê-lo.

A primeira parte do estudo foca-se na análise de SFs. Verificou-se que, quando um `fsync` falha, o SF pode marcar páginas como sujas ou limpas, manter o conteúdo antigo ou reverter o ficheiro para um estado anterior e que a indicação do erro pode ser imediata ou apenas pode ocorrer na próxima chamada ao `fsync`.

Para avaliar a resposta das aplicações às falhas de `fsync`, dado o comportamento verificado dos SF, foi desenvolvido o **CuttleFS**, um sistema de ficheiros FUSE.

O CuttleFS possui a sua própria cache de páginas em espaço de utilizador. Quando a aplicação solicita um `fsync`, os dados da cache do CuttleFS são sincronizadas com o ficheiro de sistemas situado na camada abaixo. Este possui dois modos de operação:

- Modo de rastreio: rastreia as escritas e quais blocos são eventualmente escritos para disco.
- Modo de falta: pode ser configurado para falhar o *n*-ésimo `write` de um setor ou bloco para um ficheiro em particular. Imita os comportamentos dos SF estudados. Adicionalmente, pode substituir todas ou algumas páginas limpas da cache.

Para cada aplicação, foi desenvolvido uma carga de trabalho simples (inserção ou atualização de dados) e o CuttleFS foi usado no modo de rastreio. Analisando manualmente os *traces* das aplicações, foram geradas sequências de falhas e definida a reação do SF (sendo que é obrigatório fornecer uma das combinações definidas).

Algumas conclusões retiradas do estudo são que as aplicações são destinadas a executar sobre SFs específicos; que apesar de aplicarem um conjunto de estratégias para lidar com falhas de `fsync`, estas são insuficientes; e que a lógica de recuperação das aplicações acede a dados incorretos da cache.

Um dos sistemas de ficheiros FUSE para injeção de faltas que tem especial interesse para esta dissertação é o **LazyFS** [101]. O LazyFS permite a simulação de falhas de energia através da limpeza dos dados em *cache*, já que essa é uma consequência direta deste tipo de falhas. Para isto ser possível, o LazyFS possui uma *cache* própria. Assim, as chamadas ao sistema são programadas de modo a que os dados sejam guardados e obtidos da sua cache interna. Ao contrário da *cache* do SO, os dados nela contidos apenas são *flushed* com pedidos explícitos da aplicação, tais como o `fsync()` e o `fdatasync()`.

As faltas são injetadas através da comunicação com um processo dedicado à escuta de comandos de injeção de faltas. Estes comandos são de dois tipos. Um deles é um pedido simples de limpeza de *cache*, sem qualquer condição associada. Já o outro, permite a especificação da limpeza de *cache* depois ou antes da ocorrência de uma dada chamada ao sistema para um dado ficheiro (ou ficheiros, em caso de chamadas como o `rename` e `link`). Neste último caso, a *cache* é limpa como consequência da interrupção da execução do LazyFS, de modo a impedir que a aplicação execute as operações que iria executar a seguir.

O LazyFS é um injetor de faltas que possui vários pontos positivos, sendo um deles a capacidade de reproduzir testes, já que a *cache* é mantida pelo próprio, sendo a sua gestão é determinística. Para além disso, é possível especificar pontos concretos de injeção de faltas, que não dependem de tempo, mas sim de eventos.

Uma outra vantagem do LazyFS é o facto de disponibilizar informação acerca dos dados que estão em risco, dando a conhecer ao utilizador os blocos de dados dos ficheiros, com as respetivas posições, que não foram sincronizados. Tais informações facilitam bastante na tarefa de *debugging*.

2.3.4 Discussão

Os sistemas de ficheiros FUSE desenvolvidos para injeção de faltas, mencionados anteriormente, constituem uma abordagem caixa negra, já que são independentes da aplicação alvo. O único requisito é que as aplicações sejam compatíveis com a interface POSIX. Isto contribui não só para a portabilidade do injetor de faltas, como torna a tarefa de testar o sistema mais fácil para utilizador, já que este não necessita de anotar o sistema, como acontece com o EXPLODE, em que é necessário definir pontos em que a execução pode ser diferente.

Como mencionado na Secção 2.3.2, existem algumas ferramentas que procuram encontrar automaticamente os vários estados de falhas. Esta estratégia opõe-se à dos injetores de faltas que injetam uma falta de cada vez. O ALICE, por exemplo, explora as vulnerabilidades e verifica se as mesmas levam à falha do sistema. Esta procura extensiva é vantajosa para alguns propósitos, como, por exemplo, na tentativa de encontrar problemas nunca antes reportados. Contudo, existe interesse em disponibilizar uma ferramenta capaz de reproduzir problemas já conhecidos. Este tipo de ferramenta até pode ser complementar a ferramentas exploratórias. É útil ter uma ferramenta que possa reproduzir *bugs* reportados por utilizadores, por exemplo, e que possa ser utilizada em cadeias de testes automáticos. O LazyFS, por exemplo, está integrado com o Jepsen, sendo que o Jepsen explora, de forma aleatória, os vários cenários de falta e o LazyFS é utilizado para injetar faltas. É de notar, no entanto, que nem todos os injetores de faltas permitem reproduzir testes realizados, como, por exemplo, o UnreliableFS, visto que as faltas, nesta ferramenta, são injetadas com uma dada probabilidade, ou seja, não são definidos pontos específicos da execução onde a falta será injetada.

O CuttleFS, tal como o LazyFS, possui uma cache interna, de modo a ter um controlo determinístico dos dados que são persistidos. O foco destas duas ferramentas é, no entanto, bastante diferente. O CuttleFS verifica de que forma as aplicações reagem à falha do `fsync`, ou seja, não estuda a correta utilização do `fsync`. O LazyFS, por sua vez, assume que as camadas abaixo da camada aplicacional funcionam corretamente e estuda o problema das aplicações não persistirem explicitamente os seus dados.

Pretende-se expandir o tipo de faltas suportadas em ferramentas como o LazyFS, para permitir a reprodução de problemas que tipicamente são difíceis de reproduzir, como é o caso da reordenação e atomicidade de escritas. Como é possível verificar na Tabela 1, todas as ferramentas que abordam estes dois problemas são de âmbito exploratório.

A ferramenta desenvolvida avançará o estado da arte da validação da confiabilidade de sistemas de armazenamento, principalmente no que diz respeito à reprodutibilidade dos testes, mas também

Tabela 1: Resumo do trabalho relacionado. A coluna "sistemas testados"indica em que tipo de sistemas as ferramentas foram efetivamente testadas. As ferramentas em que não foram encontrados dados acerca de sistemas reais testados têm uma indicação de "não aplicável". Uma injeção ser simulada indica que as faltas não são injetadas em tempo de execução, mas os sistemas são executados, enquanto que a injeção ser estática significa que os sistemas não são executados.

Ferramenta	**Alvo de estudo**	**Sistemas testados**	**Modelo de faltas**				**Propósito**	**Injeção**
			Atomicidade	*Reordenação*	*Durabilidade*	*Outro*		
CrashMonkey [56]	F	F	X	X	X		E	S
Ferrite [46]	A	—	X	X	X		E	S
Zheng *et al.*	A	L			X		E	S
ALICE [39]	A	LD	X	X	X		E	S
PACE [44]	A	D	X	X	X		E	S
FATE e DESTINI [30]	A	D				X	E	T
EXPLODE [25]	FA	F	X	X	X	X	E	S
ELEVEN82 [50]	A	LD	X	X	X		E	E
C^3 [49]	A	O	X	X	X		E	S
CORDS [55]	A	D				X	RI	T
UnreliableFS [84]	A	—				X	I	T
CharydbeFS [48]	A	D				X	I	T
PetardFS [105]	A	U				X	RI	T
CuttleFS [63]	A	L				X	RI	T
LazyFS [101]	A	LD			X		RI	T

Alvo de estudo
A - Aplicação
F - Sistema de Ficheiros
— Não aplicável

Sistemas testados
F - Sistemas de ficheiros
L - Armazenamento local
D - Armazenamento Distribuído
O - Outro

Modelo de faltas
X - Falta abrangida pela ferramenta

Propósito
E - Exploração
I - Injeção única
R - Reprodução

Injeção
S - Simulada
T - Tempo de execução
E - Estática

relativamente a aspetos como portabilidade e facilidade de utilização. Excluindo o ALICE, que é capaz de indicar as linhas de origem das vulnerabilidades, e o LazyFS, que fornece relatórios acerca dos dados em risco em risco de serem perdidos, nenhuma das ferramentas mencionadas se preocupa na questão de fornecer *output* informativo ao utilizador, de modo a que este possa perceber melhor a causa dos comportamentos incorretos da aplicação. A nossa solução tem como objetivo ajudar o programador na tarefa de *debugging*.

Capítulo 3
Motivação

Este estudo pretende motivar uma melhor compreensão do impacto da perda total ou parcial dos dados e de que modo é que os responsáveis pelas aplicações conseguem dar resposta a estes problemas, já que podem resultar de comportamentos difíceis de reproduzir dos SFs.

O primeiro passo desta dissertação consistiu na análise de um conjunto de erros ou incorreções conhecidos em aplicações amplamente utilizadas. Estes *bugs* foram encontrados quer nos repositórios das aplicações, tendo sido reportados por utilizadores, quer em artigos científicos [39, 44].

3.1 Metodologia

O objetivo deste estudo é analisar *bugs* reportados nos repositórios das aplicações e responder ao seguinte conjunto de questões:

- Como é que estes *bugs* foram encontrados e reportados?
- Qual é a sua causa?
- Qual é o seu impacto na aplicação?
- Como é que os *bugs* foram reproduzidos e corrigidos?

A intenção deste estudo é mostrar que existem problemas decorrentes da falta de `fsyncs` e da falta de estratégias mais complexas para impedir escritas parciais. Pretende também mostrar que em alguns casos, é difícil corrigir estes problemas devido à falta de meios para reproduzi-los. O objetivo é colecionar alguns casos de estudos e analisá-los ao pormenor. Logo, a realização de um estudo compreensivo e a exploração de outro tipo de faltas é ortogonal ao propósito deste estudo.

As aplicações consideradas neste estudo foram as seguintes:

LevelDB. O LevelDB é uma biblioteca que disponibiliza um sistema de armazenamento chave-valor persistente. O LevelDB possui ficheiros de log que guardam o conjunto de atualizações recentes. Quando

o log atinge um determinado tamanho (por omissão, 4MB), é convertido numa tabela ordenada (segundo as chaves) e um novo log é criado. As tabelas ordenadas são organizadas em vários níveis e são continuamente agrupadas em níveis superiores através de um processo de compactação.

O LevelDB possui também um ficheiro `MANIFEST` com metadados importantes, tais como dados sobre as tabelas ordenadas, e um ficheiro `CURRENT` que contém o nome do ficheiro `MANIFEST` mais recente.

As escritas dos pares chave-valor podem ser realizadas de forma síncrona ou assíncrona. Por omissão, as escritas são assíncronas, ou seja, a base de dados dá como concluída uma escrita quando esta é realizada apenas para cache.

etcd. O etcd é uma *key-value store* distribuída com coerência forte. O algoritmo de acordo distribuído utilizado pelo etcd é o Raft [118]. Este algoritmo permite que múltiplos servidores estabeleçam entre si um acordo distribuído, mesmo perante problemas como partições de rede, falhas de servidores, entre outros. Uma operação no Raft e, consequentemente, no etcd, é dada como concluída quando houver um acordo entre os servidos no sistema distribuído, o que implica que foi aplicada e persistida no sistema.

O etcd utiliza WAL e *snapshotting* para coordenar operações entre os membros do sistema e para recuperar após uma falha. Cada ficheiro de WAL é um conjunto de entradas e cada entrada contém um *checksum*, um tipo e um *payload* de dados. Quando o log atinge um determinado tamanho (64 MB), um novo log é criado.

PebblesDB. O PebblesDB é um sistema de armazenamento chave-valor que tem por base da sua implementação uma variante do LevelDB, o HyperLevelDB. O HyperLevelDB possui melhor paralelismo e taxa de transferência de escritas durante a fase de compactação do que o LevelDB. Já o PebblesDB resulta, essencialmente, da alteração da estrutura de dados que é utilizada no HyperLevelDB. A estrutura utilizada, denominada *Fragmented Log-Structured Merge Tree* (FLSM), permite atingir um taxa de transferência de escritas superior e uma menor amplificação de escritas sem comprometer o taxa de transferência de leituras.

3.2 Falta de *fsync*

O primeiro e mais simples cenário de *bug* é quando falta um `fsync`, que é necessário para a persistência explícita os dados. Apesar de existirem mecanismos para impedir que os dados fiquem eternamente em memória, como a existência de *threads* que em segundo plano realizam `fsync` (como mencionado na Secção 2.2.2), é crucial que as aplicações não dependam de terceiros para persistir os seus dados, pois, por exemplo, a frequência com que esta persistência é realizada pode não ser suficiente para garantir a

crash consistency da aplicação.

Um exemplo deste problema foi encontrado no LevelDB v1.12 [1] e será descrito de seguida. O problema reportado pode ser ou não um *bug*, pois não se compreende exatamente quais as garantias dadas pelo LevelDB e se o comportamento descrito é esperado. No entanto, não deixa de ser um caso interessante de ser estudado.

O utilizador refere a possibilidade de faltar um `fsync` antes da compactação de um ficheiro de log. No LevelDB, quando um log atinge um determinado tamanho, um novo ficheiro de log é criado e é nesse novo log que os novos pares chave-valor serão escritos. Um aspeto importante acerca das escritas no LevelDB é que, no modo assíncrono, as escritas não são seguidas de `fsync`, ao contrário do que acontece nas escritas síncronas.

Assim, se uma falha de energia ocorrer no momento certo, é possível que o conteúdo do log antigo não tenha sido persistido, mas o conteúdo do novo log tenha sido. Para este cenário acontecer basta que tenha ocorrido uma escrita síncrona para o novo ficheiro de log. Deste modo, as escritas assíncronas para o novo ficheiro de log serão persistidas, enquanto que as escritas assíncronas para o ficheiro de log anterior foram perdidas e são irrecuperáveis. Este comportamento pode ser surpreendente para vários utilizadores, embora possa não constituir exatamente um *bug*.

Os passos para reprodução deste comportamento do LevelDB, segundo o utilizador, consistem em:

1. Alterar o código do LevelDB, introduzindo um `sleep()` na *thread* em segundo plano responsável por realizar a compactação.

2. Criar uma base de dados e executar uma carga de trabalho em que são realizadas escritas assíncronas até ser criado um novo log e depois é realizada uma escrita síncrona no novo ficheiro de log.

3. Após a execução da carga de trabalho, nos 5 segundos seguintes, desligar cabo de alimentação do computador. Após reiniciar o computador, ler todos os pares chave-valor da base de dados.

Constata-se, portanto, que de modo a reproduzir este *bug*, é necessário não só alterar o código fonte da aplicação, como provocar a nível do *hardware* uma falha de energia.

Este problema ainda se encontra assinalado como por resolver no repositório do GitHub do LevelDB.

[1] *Possible bug: Missing a fsync() on the ".log" file before compaction* - https://github.com/google/leveldb/issues/193

3.3 Reordenação de escritas

O segundo cenário de falta é um pouco mais complexo, pois implica uma maior preocupação no momento em que o `fsync` é invocado. É importante ter em consideração que as escritas podem ser reordenadas, isto é, blocos de dados atualizados mais tarde do que outros podem ser escritos para disco primeiro. Quando se pretende impor que uma escrita `A` seja realizada antes de uma escrita `B`, é necessário invocar um `fsync` após escrever `A`. Como é possível observar na Figura 3, se não for realizado um `fsync` após a escrita `A`, é possível que a escrita `B` seja realizada para disco primeiro.

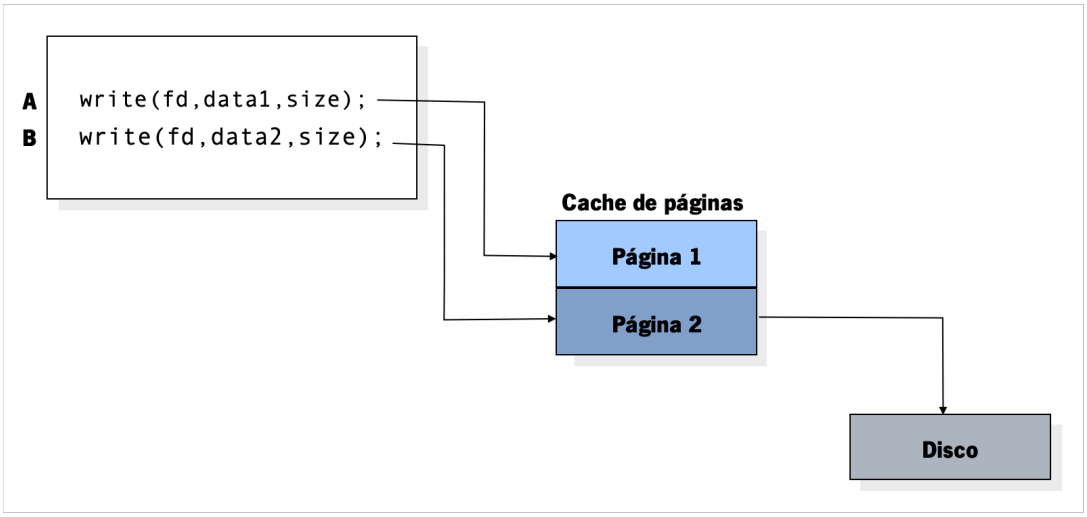


Figura 3: Reordenação de escritas.

Um dos *bugs* encontrados que cai nesta categoria de faltas ocorreu no LevelDB v1.15 [2]. Segundo o que foi relatado no problema reportado, é possível que o LevelDB contenha valores parciais nas bases de dados após a ocorrência de uma falha de energia. Tal acontece apenas quando a base de dados é configurada com o parâmetro de `paranoid_checks` definido para o valor `true`. Na documentação é referido que este parâmetro tem, por omissão, o valor `false`, mas que pode ser atribuído o valor `true` caso se pretenda que um erro seja retornado quando é detetada corrupção na base de dados. Quando uma corrupção ocorre, é sugerida a utilização de uma função denominada `RepairDB`, para recuperar, passando a citar, "o máximo de dados possível". Não é claro o que esta frase significa, mas é expectável que não signifique recuperar valores parciais.

Os efeitos deste *bug* são graves: a corrupção silenciosa dos dados. A utilização dos mecanismos fornecidos pela aplicação para recuperar de corrupções acaba por trazer consequências mais nefastas.

[2] *Possible bug: After a power-loss, LevelDB does not discard partially-flushed transactions* - https://github.com/google/leveldb/issues/251

É possível que este problema ocorra devido à reordenação de escritas. O ficheiro de log pode não ser persistido de forma contígua, levando a que, na sequência de uma falha de energia, o ficheiro possa conter dados aleatórios ou anteriores intercalados com dados corretos. O que o `RepairDB` faz é, essencialmente, descartar os dados incorretos e ficar com o resto do valor.

O problema ainda se encontra aberto no repositório do LevelDB e alguns comentários apontam para a recorrência do problema em versões posteriores, não existindo uma indicação clara de resolução.

Possivelmente, este problema foi esquecido devido à dificuldade de reprodução do mesmo. Estes foram os passos enumerados pelo utilizador para reproduzir o *bug*:

1. Utilizar uma partição com o ext3 no modo `writeback` para que nenhum outro processo interfira com o momento exato em que o *bug* ocorre.
2. Adicionar um `fdatasync` numa determinada zona do código do LevelDB.
3. Inserir um par chave-valor com um valor com uns milhares de carateres e fazer um ciclo infinito.
4. Esperar 5 segundos e cortar a energia antes de passarem 25 segundos.
5. Verificar o erro de corrupção e correr o `RepairDB`.

Como é possível perceber, o *bug* ocorre num intervalo temporal muito específico, tornando a tarefa de manualmente provocar uma falha de energia complexa e extremamente propensa a erros. Se o momento do corte de energia não for conseguido, todo o processo de repetir a experiência é temporalmente dispendioso.

Outra grande desvantagem deste processo é a necessidade da parte do utilizador de perceber o código da aplicação e alterá-lo de modo a conseguir reproduzir o *bug*. Idealmente, esse nível de compreensão do funcionamento interno do sistema não seria necessário.

Com este exemplo é possível perceber em termos práticos uma das potenciais causas da reordenação de páginas em aplicações, que resulta do desconhecimento, por parte dos programadores, deste tipo de comportamentos do sistema de ficheiros. É também possível assistir à dificuldade de reprodução deste tipo de problemas, já que é necessário um conjunto de artimanhas manuais para o fazer.

3.4 Tamanho das escritas

É comum agrupar múltiplos `writes` num único, na tentativa de melhorar o desempenho, já que isso implica a execução de menos chamadas ao sistema. Erroneamente, pode pensar-se que esta estratégia

é uma possível solução para o problema anterior. No entanto, ao agrupar várias escritas, continua-se a incorrer no problema anterior, pois a escrita será dividida em blocos e esses blocos podem ser persistidos fora de ordem pela cache de páginas.

Um *bug* no etcd resultante deste tipo de faltas foi identificado e reportado no artigo "*Correlated Crash vulnerabilities*" de Alagappan *et al.* [44], previamente mencionado na Secção 2.3. Este *bug* tem como consequência a indisponibilidade do servidor onde ocorre, pois o `etcd` é capaz de detetar que houve corrupção de dados, já que utiliza *checksums*, no entanto não adota estratégias de recuperação. Segundo o artigo, o `etcd` realiza escritas para o ficheiro de WAL, que são seguidas de `fsync`. No entanto, a atomicidade com que as escritas são realizadas para disco tem impacto na aplicação. Assim, embora não existam escritas consecutivas para um ficheiro, uma única escrita pode dar origem a problemas.

Foi possível encontrar no repositório do `etcd` um problema reportado por um utilizador que pode corresponder a esta vulnerabilidade detetada pelo PACE[3]. Analisando a resposta dada pelos responsáveis do etcd[4], verifica-se que o problema ficou por resolver devido à falta de meios de reprodução. Constata-se, portanto, que existe uma necessidade real de meios para reproduzir estes problemas reportados pelos utilizadores para uma resolução efetiva dos mesmos.

3.5 Discussão

Os problemas encontrados anteriormente conduziram a algumas conclusões preliminares e a algumas questões que esta dissertação pretende responder:

- **Questão 1**. A utilização de `fsync` não é trivial, nomeadamente quando se quer evitar a sua utilização com o propósito de obter melhor desempenho. Será um padrão comum entre aplicações evitar ou desconhecer a necessidade da sua utilização?

- **Questão 2**. Não é correto assumir que os dados que não foram sincronizados com `fsync` serão perdidos na totalidade, porque o sistema de ficheiros pode fazer *flush* de páginas da cache. Existem sistemas que apesar de utilizarem um log transacional (*e.g.*, Apache Zookeeper) não realizam `fsync` a cada escrita para o log, o que pode conduzir a situações de reordenação de escritas, tal como descrito no estudo efetuado com o ALICE [39]. Será que existem aplicações cientes deste problema?

[3] *ETCD data gets corrupted with error "alpb: crc mismatch"* - https://github.com/etcd-io/etcd/issues/11488

[4] https://github.com/etcd-io/etcd/issues/11488#issuecomment-570241825

- **Questão 3**. Nos casos apresentados não foi possível, por parte dos responsáveis pelas aplicações e dos utilizadores, reproduzir os *bugs* e, quando o foi, o método de reprodução exigia a compreensão e modificação do código fonte. Não existe uma forma melhor de assistir na reprodução e correção destes *bugs*?

- **Questão 4**. Apesar de não implementarem mecanismos de prevenção de escritas parciais, as aplicações podem, no entanto, implementar estratégias para a correção das mesmas (*e.g.*, não realizam escritas atómicas, mas utilizam *checksums* para detetar escritas parciais para as poderem descartar). Serão as aplicações capazes de recuperar destas escritas?

- **Questão 5**. Quais são as estratégias para corrigir cada um dos diferentes tipos de *bugs*?

Para responder a estas questões, desenvolvemos a ferramenta LazyFS+, que permite a reprodução dos vários problemas relatados previamente. Os próximos capítulos destinam-se à apresentação desta ferramenta e descrição da sua aplicação em sistemas reais.

Capítulo 4
LazyFS+

O LazyFS+ é uma ferramenta de injeção de faltas que parte da implementação e desenho do LazyFS. O LazyFS, apresentado na Secção 2.3, é um SF que permite a simulação de falhas de energia através da perda de dados em cache. A aplicação a ser testada realiza pedidos a este SF e os seus dados são persistidos apenas se esta realizar uma sincronização explícita dos dados da memória com o disco, invocando um `fsync`.

O LazyFS+ adiciona novos tipos de faltas, que correspondem a diferentes comportamentos que o SF pode ter. Mais concretamente, a solução desenvolvida nesta dissertação permite a reordenação de escritas e a configuração da sua atomicidade. Como foi exposto na Secção anterior, existem aplicações que não se protegem contra estas particularidades dos SFs o que, por conseguinte, se traduz em problemas para os utilizadores.

Os princípios de desenho do LazyFS+ são:

- **Abordagem caixa-negra.** O utilizador do LazyFS+ não necessita de conhecer os detalhes de implementação da aplicação alvo. O único requisito é que a aplicação seja compatível com a interface POSIX.

- **Perda de dados reproduzível.** O LazyFS+ possui uma cache interna, o que lhe confere controlo total sobre os dados, incluindo o momento em que são persistidos e como tal é feito. Sem que haja um pedido explícito por parte da aplicação, ou de uma falta introduzida pelo utilizador, para persistir os dados, o LazyFS+ mantém-nos unicamente em cache. Para uma mesma carga de trabalho da aplicação, *i.e.*, conjunto de operações de escritas determinístico, o LazyFS+ apresenta sempre o mesmo comportamento de falha.

- **Ajudar os programadores.** O LazyFS+ tem como objetivo ajudar os programadores a localizar e analisar as consequências da interação incorreta da aplicação com o SF. Assim, o *output* fornecido deve ser informativo e relevante, indicando os dados em risco e permitindo analisar e compreender o fluxo de operações realizadas pela aplicação.

- **Simulação de comportamentos de SFs**. O LazyFS+ baseia-se no comportamento de SFs reais. No entanto, relativamente ao trabalho relacionado, como o ALICE e o PACE, não necessita de construir um modelo para representar um determinado SF ou um SF genérico com determinadas propriedades de atomicidade e reordenação. O LazyFS+ reordena e configura a atomicidade das escritas através da injeção de faltas.

- **Integração com outras ferramentas.** O LazyFS+ pode facilmente ser utilizado em complementaridade com outras ferramentas, nomeadamente ferramentas exploratórias, pois expõe uma forma reprodutível e agnóstica à aplicação para injeção de faltas. Assim, o LazyFS+ pode servir como o módulo de injeção, enquanto que outra ferramenta solicita os pedidos de injeção conforme detete vulnerabilidades ou conforme realize testes aleatórios.

O facto do LazyFS+ seguir uma abordagem caixa-negra e permitir que a perda de dados seja reproduzível, permite oferecer resposta à terceira questão imposta no Capítulo anterior, que exprimia a dificuldade em testar e corrigir as aplicações.

4.1 Modelo de faltas

Esta dissertação foca-se no problema de *crash consistency*, tipicamente causado por falhas de energia. Estas falhas podem originar faltas tanto ao nível dos dados como dos metadados. No entanto, o foco será exclusivamente nos dados. Não serão consideradas situações em que, por exemplo, o tamanho de um ficheiro (metadados), após uma falha de energia, é maior do que os dados que ele contém. Para além disso, apesar das falhas de energia poderem ter repercussões a vários níveis da pilha armazenamento, como nos dispositivos de armazenamento ou nos SFs, consideramos que estes funcionam corretamente, uma vez que o objeto de estudo são as aplicações. Existem vários estudos ortogonais a este que se concentram noutras camadas da pilha de armazenamento [36, 54, 56].

As falhas de energia podem levar às seguintes situações:

- Perda total de dados, no caso de escritas ainda em cache não serem explicitamente persistidas.

- Perda parcial dos dados, nos casos em que existem escritas que não foram explicitamente persistidas, mas que o SF persistiu parcialmente.

A última situação decorre da atomicidade e reordenação das escritas. Se é realizada uma escrita de tamanho superior a um bloco de dados, essa escrita será dividida em escritas menores, pelo que não é suficiente executar um `fsync`. Além disso, quando existem múltiplos blocos de dados para serem

escritos em disco, a ordem em que serão persistidos é potencialmente diferente da ordem em que foram escritos para cache.

O LazyFS+ foca-se nas três falhas mencionadas:

- **Perda total de dados**

  As faltas que desencadeiam esta falha fazem parte do predecessor do LazyFS+, o LazyFS. Existem duas faltas deste tipo, descritas na Secção 4.2.4.

- **Perda de dados devido à falta de atomicidade**

  A falta que permite simular este comportamento é uma novidade introduzida pelo LazyFS+. Na Secção 4.2.5 é realizada uma descrição desta falta.

- **Perda de dados devido à reordenação**

  A falta que permite simular este comportamento, tal como a anterior, é uma novidade introduzida pelo LazyFS+. Também é na Secção 4.2.5 que é realizada uma descrição desta falta.

Importa salientar que o problema de reordenação e atomicidade pode manifestar-se noutras operações de ficheiros para além das escritas [46], mas estas não serão abordadas neste estudo.

4.2 Fluxo de operações

O LazyFS+ é um SF que expõe um conjunto de operações de manipulação de dados e metadados para as aplicações. Na pilha de armazenamento, este situa-se entre a aplicação e um *Backend* de Armazenamento (Secção 4.3.2), a camada onde os dados são escritos quando persistidos.

Todas as operações de manipulação de ficheiros são intersetadas pelo LazyFS+. Os pedidos realizados pela aplicação para criação ou alteração de dados e metadados vão refletir-se na Cache de Páginas interna do LazyFS+ (Secção 4.3.1). Ao contrário do que acontece em sistemas Linux, não existem mecanismos para persistir de forma periódica os dados que se encontram em cache. Assim, os dados apenas são persistidos com pedidos explícitos da aplicação, ou seja, através da invocação do `fsync` ou `fdatasync`, ou através das faltas que podem ser injetadas.

As principais operações do LazyFS+ resumem-se a leituras, escritas, operações de sincronização e a operações relacionadas com a injeção de faltas. Nesta Secção será descrita de forma pormenorizada qual seu o fluxo.

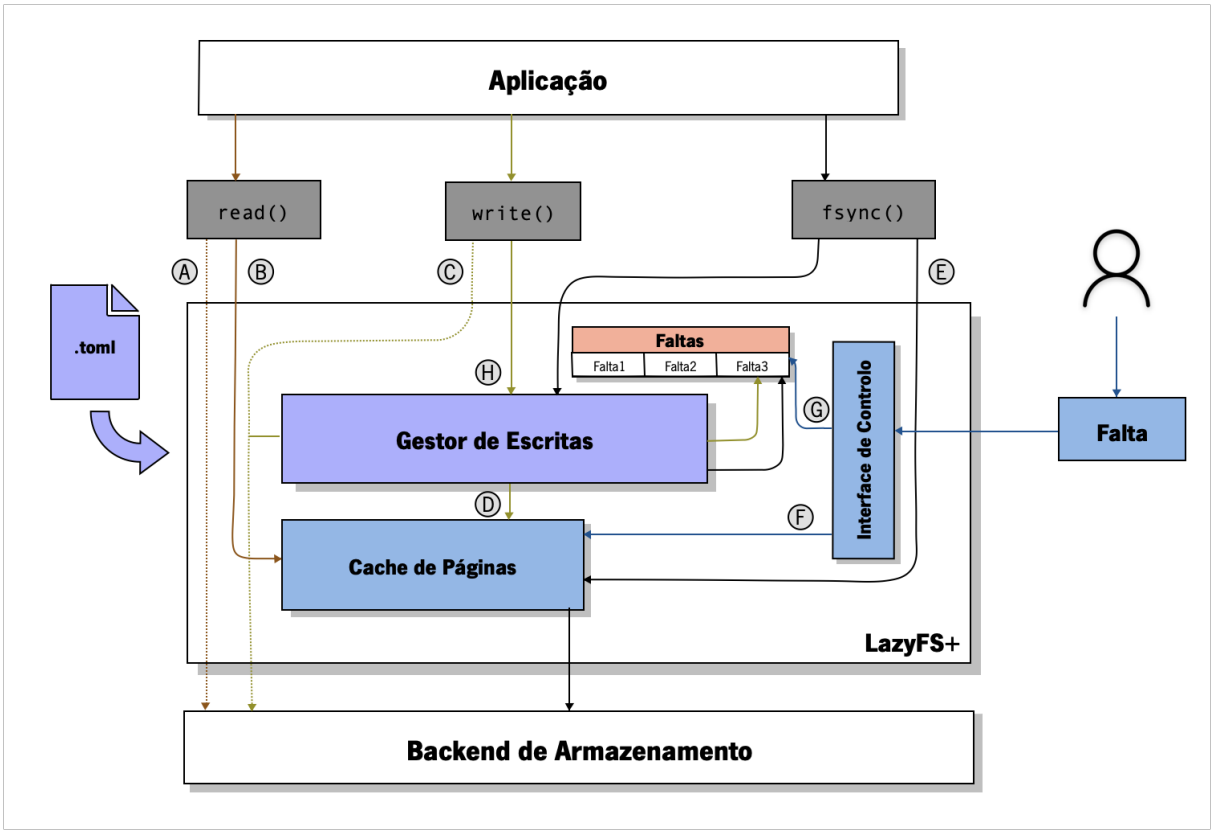


Figura 4: Arquitetura do LazyFS+ e fluxo de operações. Os componentes a violeta foram modificados e/ou adicionados à arquitetura base do LazyFS.

4.2.1 Leitura

Quando o LazyFS+ recebe um `read()`, divide o pedido em blocos, conforme o tamanho configurado. Tal como acontece na cache de páginas do Linux, estes blocos podem ser encontrados, ou não, na cache do LazyFS+ e, neste último caso, os blocos têm que ser lidos do disco. Assim, no caso da cache de páginas do LazyFS+, será verificado se o bloco existe ou não em cache (Ⓑ). Caso não exista, será solicitada a sua leitura ao *Backend* de Armazenamento (Ⓐ). No entanto, ao contrário da cache de página do Linux, este bloco lido diretamente do armazenamento não será guardado em cache. Esta decisão de desenho resulta do propósito do LazyFS+, pois o objetivo é estudar dados que estão em risco de ser perdidos, ou seja, os blocos sujos da cache e não os blocos limpos.

4.2.2 Escrita

Quando o LazyFS+ recebe um `write()`, divide o pedido em blocos, conforme o tamanho configurado, tal como acontece no caso de leitura. Os blocos podem já encontrar-se em cache e, nesse caso, estes serão reescritos (Ⓓ). No caso em que o bloco não se encontre em cache, será escrito e marcado como sujo, se a cache não estiver completamente ocupada ou, caso esteja, se existir uma política de substituição de cache. Se nenhuma dessas condições se verificar, o bloco será diretamente escrito no *Backend* de Armazenamento (Ⓒ).

Além disso, as escritas, ou uma porção destas, podem ter de ser persistidas devido à existência de faltas de reordenação ou atomicidade (mais pormenores acerca de como este mecanismo funciona são dados na Secção 4.3.4) (Ⓗ). O Gestor de Escritas é responsável por persistir dados de escritas, caso seja necessário.

4.2.3 Sincronização

Quando o LazyFS+ recebe um `fsync()`, os metadados e os dados da cache interna são sincronizados com os metadados e dados do *Backend* de Armazenamento (Ⓔ). As páginas da cache relativas ao *inode* que foi sincronizado são limpas, pois pretende-se que na cache do LazyFS+ apenas existam dados que ainda não foram persistidos. No caso do `fdatasync()`, apenas os dados são sincronizados com o *Backend* de Armazenamento.

A execução do `fsync` também tem efeitos ao nível do Gestor de Escritas, pois algumas variáveis necessárias às faltas de reordenação e atomicidade são atualizadas (será dada uma explicação mais detalhada na Secção 4.3.4).

4.2.4 Faltas injetadas na Interface de Controlo

As faltas injetadas na Interface de Controlo são aquelas que são solicitadas ao LazyFS+ em tempo de execução. Estas faltas são de dois tipos:

- **Esvaziar a cache.** Quando a Interface de Controlo recebe esta falta, liberta todos as páginas da cache (Ⓕ). Esta falta não afeta a execução do LazyFS+ nem da aplicação. O utilizador pode decidir reiniciar a aplicação para permitir que esta recupere. Este comportamento é útil para integrar com outro tipo de ferramentas, como o Jepsen. O LazyFS foi integrado com esta ferramenta, sendo que é o próprio Jepsen o responsável por terminar a execução da aplicação.

- **Falha no LazyFS+.** Quando a Interface de Controlo recebe esta falta, armazena-a numa estrutura interna de faltas para que uma falha seja desencadeada na altura especificada (Ⓖ). A cada chamada ao sistema, antes e depois de esta ser executada, é verificado se existe uma falta nessa estrutura que corresponda à falha do LazyFS+ nesse momento, para o ficheiro recebido como argumento. Se a falta for encontrada, o LazyFS+ termina a sua execução de imediato.

4.2.5 Faltas configuradas

As faltas configuradas são faltas que são especificadas no ficheiro de configuração do LazyFS+ e que, portanto, são conhecidas no início da sua execução. Estas faltas são guardadas na estrutura interna de faltas do LazyFS+ (a mesma falada no ponto anterior) e podem ser de dois tipos:

Reordenação de escritas. Reordenar escritas significa que, perante um conjunto de escritas não intercaladas por `fsync`, algumas delas serão persistidas e outras não, levando a que, na perspetiva do *Backend* de Armazenamento, as escritas solicitadas pela aplicação sejam reordenadas. De modo a tornar esta situação mais clara, vamos considerar o exemplo do Listing 4.1.

```
1 write(fd,...);
2 fsync(fd,...);
3 write(fd,...);
4 read(fd,...);
5 write(fd,...);
6 write(fd,...);
7 fsync(fd,...);
```

Listing 4.1: Conjunto de chamadas ao sistema realizadas por uma aplicação onde, por questões de simplicidade, são omitidos todos os argumentos e apenas é indicada uma variável com o descritor de ficheiro.

Neste exemplo, a primeira escrita que é realizada é logo persistida. Logo após o primeiro `fsync`, são realizadas três escritas consecutivas, no sentido em que não são intercaladas com `fsync`. Este conjunto corresponde ao primeiro grupo de escritas consecutivas, para o ficheiro `fd`, desta execução. Reordenar escritas neste conjunto significa, por exemplo, persistir apenas as escritas da linha 3 e da linha 6 e forçar o término da execução do LazyFS+ após a escrita da linha 6 ser persistida.

Atomicidade de escritas. As faltas de atomicidade permitem dividir um `write()` em vários, de tamanhos configuráveis, e persistir alguns destes. Esta falta permite simular situações em que a aplicação realiza uma escrita e, devido ao tamanho da página da cache, os dados que são escritos não podem ser persistidos de uma só vez, ou seja, de forma atómica. Além de os dados não poderem ser persistidos atomicamente, estes podem ser reordenados, por isso esta falta permite dividir a escrita em várias e persistir um qualquer subconjunto desta.

4.3 Arquitetura

Na Secção anterior foram mencionados os vários componentes do LazyFS+. Nesta Secção serão fornecidos mais detalhes acerca da arquitetura do LazyFS+.

4.3.1 Cache de Páginas

A Cache de Páginas do LazyFS+ baseia-se na cache de páginas dos sistemas Linux [29]. Cada *inode* é composto por múltiplos blocos de dados e estes blocos compõem páginas na cache de páginas. Tipicamente, o tamanho do bloco é semelhante ao tamanho de uma página da cache, no entanto estes valores podem ser configurados no LazyFS+.

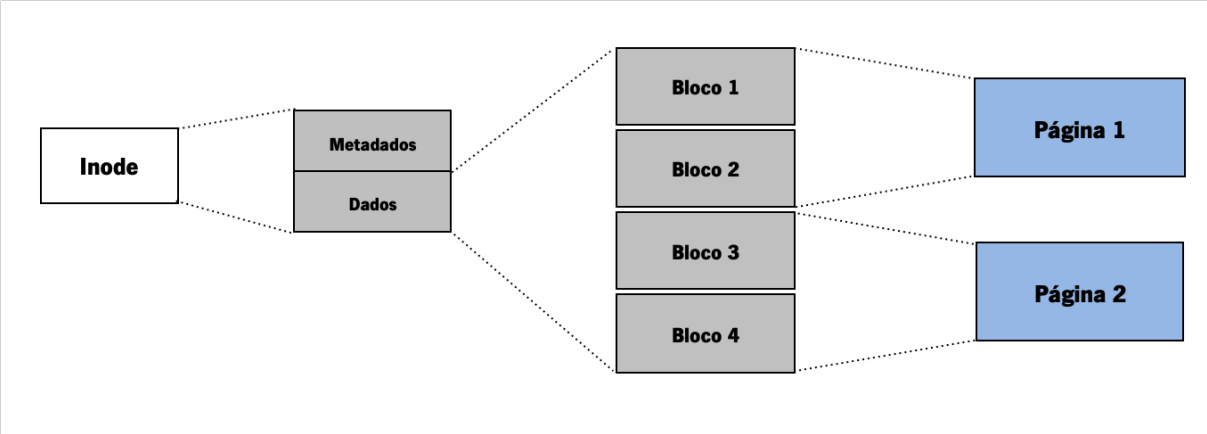


Figura 5: Estrutura da Cache de Páginas. Neste exemplo, o *inode* possui quatro blocos de dados e cada página é composta por dois desses blocos.

Cada *inode* tem a si associado metadados e dados. Os metadados contêm informações como o tamanho do ficheiro, tempo de acesso, tempo de modificação e tempo de mudança de estado e caminhos para o *inode* (diferentes nomes de ficheiros que estão ligados ao mesmo *inode*). Já os dados correspondem aos blocos sujos que se encontram na cache. Tanto os dados como os metadados são alterados na sequência da invocação de chamadas como `write()`, `read()`, `rename()`, entre outras.

Existe a possibilidade de a cache de páginas ficar sem páginas disponíveis. O LazyFS+ disponibiliza o *Least Recently Used* (LRU) como política de substituição de cache, embora a implementação desta política seja dispensável para este estudo. Ao introduzir a possibilidade da cache realizar uma sincronização das suas páginas através de alguma política, passam a existir dados que não foram sincronizados explicitamente pela aplicação que são persistidos, o que se desvia do propósito deste estudo: os dados que são essenciais à aplicação devem ser da sua responsabilidade e não se deve confiar noutros mecanismos externos para a sua persistência. Configurando o tamanho da cache do LazyFS+, é possível evitar este problema, pois as aplicações podem escolher um tamanho suficientemente grande para que os dados sejam guardados na sua totalidade em cache.

4.3.2 *Backend* de Armazenamento

O LazyFS+ situa-se acima de um sistema de armazenamento, que é o local onde os dados são guardados de forma persistente. Quando um `fsync` ou `fdatasync` é invocado, os blocos sujos da cache de páginas são escritos para esta camada de armazenamento, que pode, inclusive, ser outro SF. Neste caso, embora os dados não tenham efetivamente sido escritos em disco, caso o LazyFS+ falhe, os dados existirão na cache do SO. Só no caso de o próprio sistema falhar é que os dados serão perdidos, mas o

LazyFS+ assume que as camadas abaixo funcionam corretamente, já que o seu propósito é estudar as aplicações.

4.3.3 Interface de Controlo

O LazyFS+ possui uma interface que expõe um conjunto de métodos que permitem a injeção de faltas e a obtenção de informação sobre a cache. A Interface de Controlo é um componente que possui um *listener*, que fica à escuta de pedidos dos utilizadores, ou de outros programas, que utilizem estes métodos disponibilizados.

Este *listener* recebe e processa diferentes tipos de pedidos:

- **Eliminação dos dados em cache.** O utilizador pode solicitar o esvaziamento da cache. Assim, todos os blocos de todos os *inodes* são eliminados. Este pedido não tem qualquer efeito no LazyFS+, apenas na cache.

- **Persistir os dados em cache.** O utilizador também pode ter interesse em fazer uma sincronização de todos os blocos sujos que se encontram em cache. Este pedido apenas tem impacto ao nível da cache e do *Backend* de Armazenamento.

- **Obtenção de informação sobre os dados.** É possível solicitar a consulta do nível de ocupação da cache, que é útil para as aplicações perceberem o tamanho de cache que devem configurar para determinados testes. Também é possível obter relatórios acerca dos blocos que se encontram sujos na cache.

- **Falha do SF**. Existem vários problemas de *crash consistency* que apenas são expostos em janelas temporais muito pequenas. Um desses exemplos é o *bug* do LevelDB [1] reproduzido pelo LazyFS. Neste caso, um problema ocorre caso uma falha de energia impeça a execução das subsequentes operações solicitadas pela aplicação. Esta situação é substancialmente diferente do caso em que apenas se está a perder os dados em cache. Impedir que a aplicação realize mais chamadas ao SF é uma representação mais fiel de uma falha de energia, já que na sua sequência efetivamente tudo falha: o sistema operativo, o SF e a aplicação.

  De modo a possibilitar a reprodução deste tipo de problemas, um dos pedidos que pode ser realizado ao LazyFS+ é que este falhe em determinadas circunstâncias. Estas circunstâncias são, na prática, a ocorrência de determinadas operações, isto é, é possível pedir ao LazyFS+ para falhar

[1] *Possible bug: Missing a fsync() or msync() call after creating MANIFEST-000001* - https://github.com/google/leveldb/issues/189

após ou antes a ocorrência de de uma dada chamada ao sistema para um determinado ficheiro. O LazyFS+ é terminado abruptamente, mas o *backend* permanece intacto, de modo a não afetar os dados já persistidos.

4.3.4 Gestor de Escritas

O Gestor de Escritas é o componente responsável pelas faltas de reordenação e atomicidade de escritas e é uma das adições do LazyFS+ ao LazyFS.

No final do processamento de uma escrita, o LazyFS+ verifica se é necessário persisti-la ou dividi-la e persistir algumas porções. O primeiro critério para uma escrita ser persistida ou dividida é existir uma falta de reordenação ou atomicidade para esse ficheiro. Logo, será verificado na estrutura de faltas se existe uma falta de reordenação ou atomicidade para o ficheiro onde a escrita foi realizada.

No caso de existir uma falta reordenação, podem ser atualizados dois contadores distintos:

- Contador de grupo - contador para grupos de escritas consecutivas para um dado ficheiro. É reiniciado quando é invocado um `fsync` ou `fdatasync`.
- Contador de escritas dentro de um grupo - quando é detetado um grupo de escritas consecutivas, existe um contador para cada escrita dentro do grupo. No exemplo do Listing 4.1, a escrita da sexta linha é a terceira escrita dentro do grupo.

Assim, uma escrita só é persistida se a falta for definida para o grupo em que esta se insere e, assumindo que ela ocupa a posição `n` no grupo, se a falta for definida para persistir a `n`-ésima escrita.

Já para as faltas de atomicidade, é necessário apenas um contador de escritas. O utilizador especifica a posição da escrita na execução do LazyFS+ onde ocorrerá a falta de atomicidade e, quando o contador atinge esse valor, o Gestor de Escritas irá dividir a escrita no número de porções especificado e persistir as porções que estão configuradas para ser persistidas.

Nestes dois tipos de faltas, após as escritas que estão configuradas para ser persistidas o são, o Gestor de Escritas força a falha do LazyFS+, terminando o seu processo.

Em suma, as funções deste componente incluem:

- Verificação da existência de faltas de escrita para os ficheiros.
- Divisão de escritas em porções mais pequenas e desencadeamento da persistência de algumas dessas partes.
- Contagem e reordenação de escritas consecutivas (não intercaladas por `fsyncs`).

- Desencadeamento de falhas no LazyFS+.

4.4 Implementação

O LazyFS+ foi implementado adicionando cerca de 400 linhas de código C++ ao LazyFS. A solução é um SF em espaço de utilizador desenvolvido com o FUSE.

O FUSE é uma ferramenta para sistemas operativos UNIX (e semelhantes), que permite a implementação de SFs em espaço de utilizador, como alternativa à programação do *kernel*, que possui um grau elevado de complexidade. Esta ferramenta possui dois componentes principais, sendo que um se encontra em espaço de utilizador, a biblioteca `libfuse`, e outro componente situa-se no *kernel*. A biblioteca disponibiliza uma interface de comunicação com o componente do *kernel*. Um sistema de ficheiros FUSE é uma aplicação que utiliza a biblioteca `libfuse` para implementar as chamadas ao sistema. Quando uma aplicação invoca uma chamada ao sistema para um ficheiro, o módulo do *kernel* irá invocar a chamada do sistema implementada pelo sistema de ficheiros FUSE. Assim, o módulo do *kernel* funciona como uma ponte entre as aplicações e o SF implementado em espaço de utilizador.

O LazyFS+ implementa as operações que a maior parte dos sistemas de armazenamento compatíveis com a interface POSIX utilizam. Estas chamadas incluem métodos para gestão dos dados como `write()`, `read()`, `truncate()`, `fsync()` e `fdatasync()` e métodos para gestão dos metadados como `open()`, `create()`, `getattr()`, `rename()`, `link()` e `unlink()`.

Por omissão, o FUSE usa *multithreading*, de modo a permitir que múltiplos pedidos sejam servidos ao mesmo tempo. O LazyFS+ faz controlo de concorrência da atualização de metadados e procedimentos de alocação na cache utilizando *locks* de leitores-escritor, em que podem existir leituras concorrentes, mas as escritas são protegidas. Para além disso, os contadores necessários para as faltas de reordenação e atomicidade são atómicos [117], de modo a impedir que ocorram corridas.

Outro aspeto relevante de implementação é a forma como as faltas de limpeza da cache e falha do LazyFS+ são injetadas. Como já foi mencionado, o LazyFS+ possui um *listener*, que possui uma *thread* responsável por escutar esses pedidos, que são comunicados através de um pipe com nome (um ficheiro FIFO) [66]. Este tipo de faltas são assíncronas, já que é indefinido o momento em que o LazyFS+ irá receber o pedido de falta. No caso da falha do sistema, é possível perceber o momento em que o sistema injeta a falta, pois a sua consequência é o LazyFS+ falhar. No entanto, no caso da limpeza da cache, só consultando o log é que se torna possível verificar se a falta foi ou não injetada com sucesso.

Assim, para além do Gestor de Escritas, o LazyFS+ também adiciona ao LazyFS uma forma de notificar o utilizador quando uma falta de limpeza de cache é concluída, através da escrita noutro FIFO de uma

indicação da finalização da injeção da falta.

4.5 Configuração

O LazyFS+ é configurável através de um ficheiro `TOML`. Esta linguagem oferece um formato de texto legível, simples e claro, que permite, com facilidade, mapear as configurações para estruturas de dados.

É possível configurar parâmetros como o tamanho das páginas e dos blocos da cache de páginas, o caminho onde são injetadas as faltas em tempo de execução e o caminho do log produzido. Os parâmetros que permitem estas configurações são:

- `[faults]`

  Esta secção permite configurar os caminhos dos FIFOs. Um dos parâmetros necessários é o `fifo_path` que define o FIFO que comunica com a Interface de Controlo. Um parâmetro opcional é o `fifo_path_completed`, que define o FIFO onde são escritas as mensagens de conclusão de injeção das faltas de limpeza de cache.

- `[cache]`

  Esta secção permite definir as características da cache. Existem dois tipos de configuração: a simples (`cache.simple`), em que apenas é necessário configurar o tamanho total da cache, e a manual (`cache.manual`), que permite uma maior personalização. Na configuração simples, o tamanho do bloco é de 4 KiB e cada página possui um bloco apenas.

- `[filesystem]`

  Esta secção permite configurar o log do LazyFS+. Informações como a receção de um pedido de injeção de falta e desencadeamento da injeção de faltas são sempre imprimidas no ecrã. No entanto, é possível também fazer log das chamadas ao sistema processadas pelo LazyFS+. Para isso, é necessário definir o valor `true` para o parâmetro `log_all_operations`. Opcionalmente também é possível definir o caminho para um ficheiro de log.

O LazyFS+ acrescenta outro tipo de configurações, que se destinam à caracterização das faltas de reordenação e atomicidade. A decisão de definir estas faltas no início da execução do LazyFS+, em vez de as injetar em tempo de execução, tal como acontece com as faltas do LazyFS, relaciona-se com a especificidade deste tipo de faltas e a necessidade de sincronização. As faltas do LazyFS levam à perda total dos dados que se encontram em cache, ao contrário das faltas de reordenação e atomicidade. Existe interesse em injetar as faltas do LazyFS em diversos momentos da execução da aplicação, já que existe

a possibilidade de em múltiplos pontos de execução existirem dados importantes em cache. No entanto, os pedidos realizados à Interface de Controlo são assíncronos, o que significa que o momento exato em que o LazyFS reconhece a existência desta falta é indefinido.

As faltas de atomicidade, por exemplo, embora sejam altamente configuráveis e permitam testar atomicidade de escritas de 1 *byte*, para serem utilizadas de forma realista, convém configurá-las para dividir um `write` em porções de 4 KiB, pois este é o tamanho típico de uma página da cache. A frequência com que existem escritas de tamanho superior a este valor pode ser baixa. O mesmo se aplica às faltas de reordenação (nas aplicações testadas, foi possível verificar isto). Assim, existe interesse em que estas faltas sejam síncronas, já que ocorrem a momentos mais específicos da execução das aplicações. Caso fossem assíncronas, o momento da execução em que o LazyFS+ tomaria conhecimento da falta é indefinido, sendo possível que o ponto em que a falta deveria ter sido injetada já tenha passado.

A configuração do LazyFS+ relativo a estas faltas é denominado `injection` e é opcional. O parâmetro comum às faltas de reordenação e atomicidade é o `type`, que permite definir o tipo da falta a ser injetada. O tipo pode ser `reorder` ou `split_write`.

- `reorder`: Este tipo de falta é injetada quando uma sequência de chamadas ao sistema, destinada a um dado ficheiro, é executa consecutivamente sem a intervenção de um `fsync()`. Existem quatro parâmetros necessários para a definição deste tipo de falta. É necessário definir o ficheiro alvo da falta e que a operação a ser reordenada é uma escrita. Para além disso, através do parâmetro `occurrence` identificamos o grupo de escritas consecutivas em que a falta será injetada (a definição do que é este grupo foi dada na Secção 4.2). Por fim, é necessário definir quais escritas serão persistidos, para isso existe o parâmetro `persist`, que é um vetor que contém a numeração das escritas que serão persistidas.

  Na configuração do Listing 4.2 será injetada uma falta de reordenação no segundo grupo de escritas consecutivas para o ficheiro `log1` e a primeira e quarta escritas serão persistidas.

- `split_write`: Esta falta permite a divisão de um `write()` em escritas mais pequenas e persistir algumas dessas. Tal como no caso anterior, existem 4 parâmetros necessários à definição deste tipo de falta. Um dos parâmetros é o ficheiro alvo da falta, definido no campo `file`. O parâmetro `occurrence` deve ser definido para indicar a escrita onde a falta será injetada. Tal como na falta de reordenação, é necessário definir o vetor `persist` para indicar as porções da escrita que serão persistidas. Para uma escrita ser dividida, é necessário indicar como a divisão será realizada. Existem dois parâmetros que permitem fazer isso, no entanto, são mutuamente exclusivos. Uma possibilidade é definir esta divisão através do parâmetro `parts`, em que é indicado

o número de partes, com tamanho semelhante, em que a escrita será dividida. Alternativamente, é possível definir o vetor `parts_bytes` em que cada índice do vetor corresponde ao tamanho de uma porção da escrita.

Na configuração do Listing 4.2 será injetada uma falta de divisão de escritas no quinto `write()` para o ficheiro `log2`. Se o parâmetro de divisão for o `parts`, a escrita será dividida em 3 partes de tamanho semelhante e se for o parâmetro `parts_bytes` definido, a escrita será dividida numa escrita de 4096 *bytes*, outra de 3600 *bytes* e, por último, uma de 1260 *bytes*. A primeira e a terceira porções da escrita serão persistidas. Se, por acaso, a soma das partes do `parts_bytes` não for igual ao tamanho da escrita, uma mensagem de erro será imprimida a indicar que é necessário retificar os valores e nenhuma falta será injetada.

```
[faults]
fifo_path="/tmp/faults.fifo"                      #necessário
fifo_path_completed="/tmp/faults_completed.fifo" #opcional

[cache]
apply_eviction=false    #opcional

[cache.simple]          #opcional
custom_size="0.5GB"     #necessário (se [cache.simple])
blocks_per_page=1       #opcional

[file system]               #opcional
log_all_operations=false    #opcional
logfile="/tmp/lazyfs.log"   #opcional

[[injection]]           #opcional
type="reorder"          #necessário (se [[injection]])
op="write"              #necessário (se [[injection]] e type="reorder")
file="log1"             #necessário (se [[injection]] e type="reorder")
persist=[1,4]           #necessário (se [[injection]] e type="reorder")
ocurrence=2             #necessário (se [[injection]] e type="reorder")

[[injection]]          #opcional
type="split_write"     #necessário (se [[injection]])
file="log2"            #necessário (se [[injection]] e type="split_write")
ocurrence=5            #necessário (se [[injection]] e type="split_write")
parts=3                #necessário (se [[injection]] e type="split_write")
#parts_bytes=[4096,3600,1260]
persist=[1,3]          #necessário (se [[injection]] e type="split_write")
```

Listing 4.2: Exemplo de configuração do LazyFS+.

4.6 Comandos da Interface de Controlo

Como foi dito anteriormente, o LazyFS+ possui uma interface que permite a comunicação, em tempo de execução, entre o LazyFS+ e o utilizador. Esta comunicação é realizada através de um FIFO e os pedidos realizados podem ser:

- **Limpeza da cache**. Para injetar esta falta é necessário escrever no FIFO `lazyfs::clear-cache`.
- **Falha no LazyFS+**. A *string* para injeção desta falta tem o formato `lazyfs::crash::timing=I::op=O::from_rgx=F::to_rgx=T`, sendo que o momento de injeção `I` pode ser antes (`before`) ou depois (`after`) uma dada chamada ao sistema `O`. Adicionalmente, quando a chamada ao sistema `O` envolve dois caminhos, como no caso de um `rename`, define-se o valor desses caminhos como expressões regulares em `from_rgx` e `to_rgx`. Já no caso em que só um caminho é necessário, define-se esse valor em `from_rgx`.
- **Persistir dados em cache**. Para que os blocos de dados em cache sejam persistidos pode escrever-se no FIFO `lazyfs::cache-checkpoint`.
- **Ocupação da cache**. Para obter a informação da percentagem de páginas da cache ocupadas pode usar-se o comando `lazyfs::display-cache-usage`.
- **Blocos sujos**. O comando `lazyfs::unsynced-data-report` itera sobre os blocos da cache e imprime no ecrã informação sobre o ficheiro a que pertencem, bem como as posições não persistidas.

4.7 *Output*

O *output* produzido por uma ferramenta de *debugging* é crucial para a compreensão e correção dos erros do sistema alvo de teste. Um dos princípios de desenho do LazyFS+ consiste em ajudar os programadores e utilizadores nestas tarefas.

Nos Listings 4.3, 4.4, 4.5, 4.6, 4.7 e 4.8 podemos ver o tipo de informação reportada pelo LazyFS+.

```
[2023-10-24 16:32:30.463] [global] [info] [lazyfs.faults.worker]: received
    'lazyfs::display-cache-usage'
[2023-10-24 16:32:30.463] [global] [warning] [lazyfs.cmds]: cache usage (%
    pages) is 0.002288818359375%
```

Listing 4.3: Informação reportada quando é enviado um pedido para consulta da ocupação da cache.

```
[2023-10-24 16:32:59.341] [global] [info] [lazyfs.faults.worker]: received
   'lazyfs::unsynced-data-report'
[2023-10-24 16:32:59.341] [global] [warning] [lazyfs.cmds]: report request
   submitted...
[2023-10-24 16:32:59.342] [global] [warning] [lazyfs.cmds]: report
   generated.
[2023-10-24 16:32:59.342] [global] [info] [lazyfs.cmds]: report: [inode
   6296299] is not fully synced, info:
[2023-10-24 16:32:59.342] [global] [info] [lazyfs.cmds]: report: [inode
   6296299] info: (block 0) to (block 2) [byte index 0 to index 8963]
[2023-10-24 16:32:59.342] [global] [info] [lazyfs.cmds]: report: [inode
   6296299] files mapped to this inode:
[2023-10-24 16:32:59.342] [global] [info] [lazyfs.cmds]: report: [inode
   6296299] => file: '/home/gsd/test-r/file2.txt'
[2023-10-24 16:32:59.342] [global] [info] [lazyfs.cmds]: report: total
   number of bytes un-fsynced: 8964 bytes.

[2023-10-24 16:32:59.342] [global] [info] [lazyfs.cmds]: report: [inode
   6291537] is not fully synced, info:
[2023-10-24 16:32:59.342] [global] [info] [lazyfs.cmds]: report: [inode
   6291537] info: (block 2) to (block 2) [byte index 0 to index 781]
[2023-10-24 16:32:59.342] [global] [info] [lazyfs.cmds]: report: [inode
   6291537] files mapped to this inode:
[2023-10-24 16:32:59.342] [global] [info] [lazyfs.cmds]: report: [inode
   6291537] => file: '/home/gsd/test-r/file1.txt'
[2023-10-24 16:32:59.342] [global] [info] [lazyfs.cmds]: report: total
   number of bytes un-fsynced: 9746 bytes.
```

Listing 4.4: Relatório fornecido pelo LazyFS+ sobre os dados em cache.

```
[2023-10-24 16:33:50.061] [global] [info] [lazyfs.faults.worker]: received
   'lazyfs::clear-cache'
[2023-10-24 16:33:50.061] [global] [warning] [lazyfs.cmds]: clear cache
   request submitted...
[2023-10-24 16:33:50.062] [global] [warning] [lazyfs.cmds]: cache is
   cleared.
```

Listing 4.5: Informação imprimida quando é recebida uma falta para limpar os dados em cache.

```
[2023-10-24 16:36:10.074] [global] [critical] Triggered fault condition (op
   =write,timing=before)
[2023-10-24 16:36:10.074] [global] [warning] [lazyfs.cmds]: report request
   submitted...
[2023-10-24 16:36:10.074] [global] [warning] [lazyfs.cmds]: report
   generated.
[2023-10-24 16:36:10.074] [global] [info] [lazyfs.cmds]: report: [inode
   6291537] is not fully synced, info:
[2023-10-24 16:36:10.074] [global] [info] [lazyfs.cmds]: report: [inode
   6291537] info: (block 0) to (block 2) [byte index 0 to index 8953]
[2023-10-24 16:36:10.074] [global] [info] [lazyfs.cmds]: report: [inode
   6291537] files mapped to this inode:
```

```
[2023-10-24 16:36:10.074] [global] [info] [lazyfs.cmds]: report: [inode
   6291537] => file: '/home/gsd/test-r/file1.txt'
[2023-10-24 16:36:10.074] [global] [info] [lazyfs.cmds]: report: total
   number of bytes un-fsynced: 8954 bytes.

[2023-10-24 16:36:10.074] [global] [critical] Killing LazyFS pid 16637!
```

Listing 4.6: Informação imprimida quando é desencadeada uma falta de falha do LazyFS+ segundo uma dada condição. É indicada a falta e é fornecido um relatório sobre os dados que não estavam persisistidos.

```
[2023-10-24 16:45:13.336] [global] [warning] [lazyfs.faults]: Write to path
    /home/gsd/test-r/file1.txt: will persist 4477 bytes from offset 4477
```

Listing 4.7: Sempre que uma falta de reordenação é ativada, o LazyFS+ dá informação sobre os dados que persiste.

```
[2023-10-24 16:48:22.457] [global] [warning] [lazyfs.faults]: Going to
   persist the write 2 for the path /home/gsd/test-r/file2.txt
```

Listing 4.8: Sempre que uma falta de atomicidade é ativada, o LazyFS+ dá informação sobre os dados que persiste.

O log produzido pelo LazyFS+ regista as chamadas ao sistema invocadas pela aplicação, indicando os seus argumentos, e o momento em que são injetadas faltas. Analisar a sequência de operações que as aplicações executam facilita a deteção de potenciais problemas. Deste modo, para além do LazyFS+, desenvolvemos um *parser* para filtrar este log, tornando-o mais legível e ajudando na tarefa de encontrar vulnerabilidades. Este *parser* permite:

- Filtrar chamadas ao sistema específicas.
- Filtrar chamadas ao sistema para um ficheiro em específico.
- Criar um grafo que permite visualizar a sequência de chamadas ao sistema filtradas.
- Agrupar chamadas ao sistema, isto é, se as mesmas chamadas ao sistema forem chamadas sequencialmente, o *parser* identifica este padrão e simplifica o *output*. Isto facilita a deteção de grupos de escritas consecutivas, por exemplo.
- Obter informação acerca da localização de escritas maiores que uma página, sendo que o valor da página é configurável.

Na Figura 6 é apresentado um exemplo de um grafo produzido pelo *parser*. Algumas chamadas ao sistema foram filtradas e é possível ver que alguns grupos foram formados, pois cada retângulo corresponde a um grupo de chamadas ao sistema e o valor em `reps` indica quantas vezes o padrão se repetiu.

Assim, por exemplo, o primeiro retângulo com o valor de `reps` a 2 traduz-se na sequência de chamadas ao sistema especificadas no Listing 4.9.

```
fsync("/home/gsd/test-r/file1.txt")
write("/home/gsd/test-r/file1.txt")
fsync("/home/gsd/test-r/file1.txt")
write("/home/gsd/test-r/file1.txt")
```

Listing 4.9: Sequência de chamadas ao sistema.

Quando uma escrita pertence a um grupo, valores como o seu tamanho e posição são omitidos, mas é possível desativar esta opção de agrupamento para visualizar tal informação.

É também possível solicitar que a informação da Figura 6 seja dada em formato textual.

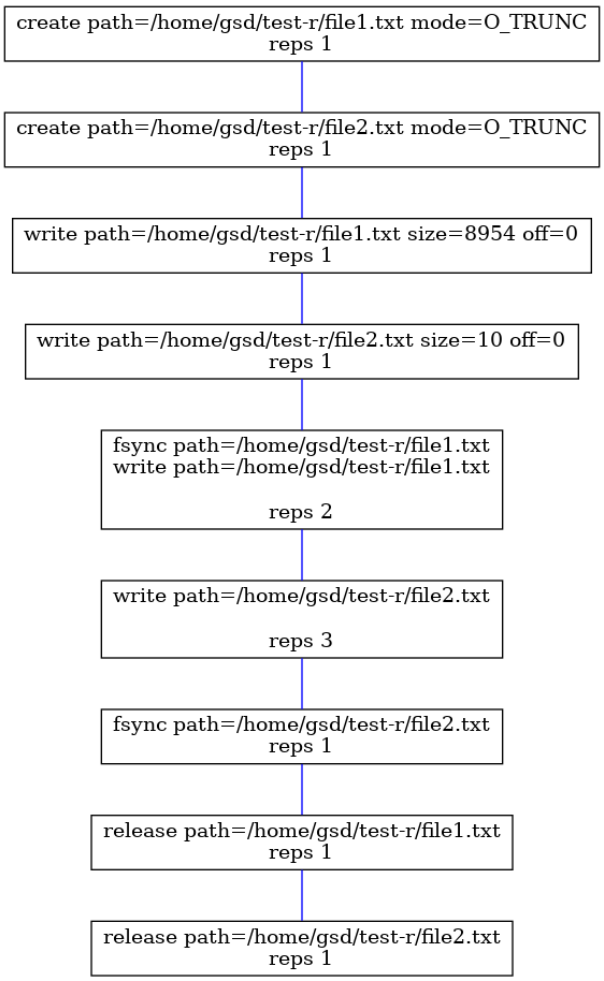


Figura 6: Exemplo de grafo produzido pelo *parser* a partir de um log do LazyFS+.

Adicionalmente, como foi mencionado, é possível solicitar ao *parser* que este identifique escritas maiores que uma página. Esta funcionalidade é útil para identificar potenciais vulnerabilidades da aplicação. O Listing 4.10 mostra um exemplo de *output*. Caso existam este tipo de escritas, a informação fornecida pelo *parser* é adequada para a aplicação ser testada sobre o LazyFS+, configurando-o com uma falta de

atomicidade de escritas. O valor `occurrence` indicado pode ser diretamente utilizado na configuração de uma falta de atomicidade, pois corresponde à posição da escrita na execução do LazyFS+. Iniciando a aplicação com o mesmo estado, configurando a falta de divisão da escrita indicada pelo *parser* e aplicando a mesma carga de trabalho, é possível verificar se a escrita é, ou não, problemática. No próximo Capítulo torna-se evidente a utilidade deste tipo de informação.

```
>  /home/gsd/test-r/file1.txt
     - size:  8954 , off:  0 , occurrence:  1
>  /home/gsd/test-r/file2.txt
     - size:  4868 , off:  4096 , occurrence:  3
```

Listing 4.10: *Output produzido pelo parser que indica escritas maiores do que o tamanho típico de uma página (4 KiB*).

Capítulo 5
Avaliação

Neste Capítulo, apresentamos o estudo experimental conduzido com o LazyFS+ para responder às questões propostas na Secção 3.5 e demonstrar que a nossa ferramenta de injeção de faltas é fundamental para a condução deste estudo. Para tal, o LazyFS+ será utilizado para reproduzir *bugs* de múltiplas aplicações em versões em que se sabia da existência de *bugs*, verificar a sua existência noutras versões e ainda, descobrir novos *bugs*.

5.1 Metodologia

Ambiente de teste. Todos os testes foram executados no mesmo tipo de máquinas, com o Ubuntu 20.04.5 LTS (Focal Fossa) como SO, arquitetura `x86_64` e versão `5.4.0-164-generic` do *kernel*. Em termos de *hardware*, as máquinas possuem 16 GiB de RAM, um `Samsung SSD 860` de 250 GiB e o modelo `Intel(R) Core(TM) i3-4170` de Unidade Central de Processamento (UCP).

Aplicações. As aplicações testadas com o propósito de reproduzir *bugs* e encontrar novos problemas com o LazyFS+ foram o LevelDB, o PebblesDB, o etcd, que já foram descritas na Secção 3.1, e o ZooKeeper. Adicionalmente, também foram realizados alguns testes com o Redis [112] e PostgreSQL [107], já que são aplicações que possuem estratégias para impedir que as faltas de reordenação e atomicidade resultem em corrupção permanente dos dados.

ZooKeeper. O ZooKeeper [79] é um serviço de coordenação distribuído para aplicações distribuídas. O ZooKeeper ajuda na coordenação entre processos distribuídos disponibilizando uma árvore hierárquica de dados partilhada, que está organizada numa forma semelhante a um SF. Esta árvore é constituída por registos de dados, denominados *znodes*, que são semelhantes a ficheiros e diretorias num SF.

A diretoria de dados do ZooKeeper contém ficheiros com cópias dos *znodes*. Todas as alterações que são realizadas aos *znodes* são adicionadas a um log transacional e, ocasionalmente, quando o log cresce para além de um determinado limite, é criado um *snapshot* do estado atual dos *znodes*. Todos os logs

existentes precedem o *snapshot* acabado de criar.

O ZooKeeper afirma ter garantias de durabilidade fortes, o que significa que garante a persistência dos dados num log antes de permitir que a operação seja dada como concluída. Em suma, este é o conjunto de garantias de coerência fornecidas [122]:

- Coerência sequencial - As atualizações dos clientes são aplicadas na ordem em que foram enviadas.
- Atomicidade - As atualizações nunca são parciais.
- Imagem única do sistema - O cliente terá a mesma visão do sistema, independentemente do servidor a que se conecta.
- Fiabilidade - Uma atualização persistirá até ser reescrita por um utilizador.
- Atualidade - A visão do cliente do sistema é a atual dentro de um determinado espaço de tempo.

Redis. O Redis [112] é uma *data structure store* que pode ser usada como uma base de dados, cache, *message broker* e *streaming engine*. Para melhorar o seu desempenho, o Redis trabalha com um *dataset* em memória. No entanto, este disponibiliza diferentes mecanismos de persistência para casos em que é necessário garantir a durabilidade dos dados. As opções de persistência incluem:

- RDB (*Redis Database*): Cria *snapshots* do *dataset* em intervalos específicos.
- AOF (*Append Only File*): Todas as escritas são registadas neste ficheiro e podem ser executadas na inicialização do servidor de modo a reconstruir o *dataset* original.
- RDB e AOF: Criação de *snapshots* e escritas para o AOF.
- Sem persistência: A persistência é completamente desativada.

PostgreSQL. O PostgreSQL [107] é uma base de dados relacional que disponibiliza propriedades ACID para as transações. Os dados e os índices associados (índices são uma forma comum de melhorar o desempenho de uma base de dados, pois permitem que o servidor encontre linhas específicas mais rapidamente) estão organizados em ficheiros distintos: um ficheiro por tabela e um ficheiro por índice. Os ficheiros estão divididos em blocos de 8 KiB.

Configurações. As configurações utilizadas foram as indicadas nos relatórios dos *bugs* e, nos casos em que foram descobertos novos *bugs*, foi utilizada a configuração padrão. No caso do Redis, foi utilizada a

opção de persistência RDB e AOF (por omissão é apenas RDB) e no caso do PostgreSQL, num dos testes foi desativada uma opção de proteção de escritas parciais.

As aplicações foram executadas tendo o LazyFS+ como o SF nas suas diretorias de dados. Como queremos que o comportamento seja determinístico, configuramos a cache com tamanho de 1 GiB, já que as cargas de trabalho utilizadas não ocupam mais espaço do que isso, e a política de substituição de cache foi desativada. O tamanho dos blocos e das páginas são de 4 .

Cargas de trabalho e injeção de faltas. Para a maioria das aplicações não foi necessário utilizar cargas de trabalho sofisticadas, nem ferramentas de avaliação para as gerar. Tirando o PostgreSQL, em todas as aplicações foram aplicadas cargas de trabalho com uma ou mais inserções ou atualizações de dados. No caso mencionado, foi utilizada uma carga de trabalho gerada por uma ferramenta de avaliação desenvolvida pela própria aplicação.

De modo a garantir a reprodutibilidade dos testes, as aplicações foram sempre iniciadas com o mesmo estado.

5.2 Estudo

O estudo realizado analisou 6 aplicações. A tabela 2 sintetiza o resultado do estudo conduzido com o LazyFS+ para cada aplicação e para cada tipo de falta. Os testes realizados com o LazyFS+ demoraram, em média, 6 segundos.

5.2.1 Falta de perda total de dados em cache

O conjunto de casos apresentados a seguir resultou da injeção das faltas que levam à perda total dos dados em cache (*i.e.* as faltas `clear-cache` e `crash`, apresentadas na Secção 4.2.4).

Perda de pares chave-valor no LevelDB e PebblesDB Ⓐ

O primeiro caso estudado é o relatado na Secção 3.2. Como foi descrito, a inexistência de um `fsync` no ficheiro com as tabelas de dados do LevelDB leva a que dados sejam perdidos. O processo de reprodução deste comportamento, realizado pelo utilizador, envolveu alterar o código da aplicação e causar uma falha de energia cortando de forma abrupta a ligação de energia do servidor.

Este caso de estudo foi reproduzido com o LazyFS+. Na versão indicada pelo utilizador, obtivemos um erro diferente do reportado. Procedemos a testar a versão mais recente do LevelDB e obtivemos a perda de múltiplos pares chave-valor indicada pelo utilizador. Também testamos o PebblesDB, uma

Tabela 2: Sumário dos testes realizados com o LazyFS+. Cada teste possui uma etiqueta que tem uma correspondência com o texto da Secção 5. As linhas a cor de laranja referem-se a testes de validação, ao contrários dos outros, que constituem problemas encontrados nas aplicações.

Falta	**Sistema**	**Etiqueta**	**Versão**	**Reprodução**	**Resultado**		
					Setup	**Impacto**	**Erro**
Perda total de dados	LevelDB	Ⓐ	1.12	○	S	U	F
			1.23	◐	S	S	—
		Ⓑ	1.12	○	S	U	C
			1.23	◐	S	U	C
	PebblesDB	Ⓐ	—	●	S	S	—
		Ⓑ	—	●	S	U	C
	ZooKeeper	Ⓒ	3.4.8	◐	C	U	I
			3.7.1	◐	C	U	F
		Ⓔ	3.4.8	●	C	I	—
	etcd	Ⓓ	2.3.0	●	S	U	F
reorder	ZooKeeper	Ⓕ	3.4.8, 3.7.1	●	C	U	M
		Ⓖ	3.4.8	○	C	U	K
			3.7.1	◐	C	U	K
	etcd	Ⓗ	2.3.0	○	S	U	K
	LevelDB	Ⓘ	1.15	○	S	S	C
			1.12	◐	S	S	C
	Redis	Ⓜ	7.2.3	—	S	U	C
split_write	etcd	Ⓙ	2.3.0	○	S	U	K
		Ⓚ	3.4.25, 3.5, 3.6	●	SC	U	I
			3.4.25, 3.5, 3.6	●	SC	U	O
			3.4.25, 3.5, 3.6	●	SC	U	F
	PebblesDB	Ⓛ	—	●	S	S	C
	PostgreSQL	Ⓝ	15.2	—	S	S	C

Reprodução
○ Conhecido
◐ Conhecido noutra versão
● Novo
— Não aplicável

Setup
C - *Cluster*
S - *Single Node*

Impacto
U - Indisponibilidade
I - Inconsistência
S - Perda de dados silenciosa

Erro
C - Corrupção em ficheiro
M - Incompatibilidade de número mágico
K - Incompatibilidade de *checksum*
I - Base de dados inválida
F - Ficheiro em falta ou formatação incorreta
O - Outro

vez que tem como base de implementação o LevelDB, e verificamos que este comportamento de perda também ocorre.

Estratégia de injeção. Tal como foi sugerido pelo utilizador que relatou este comportamento do LevelDB, foi criada uma carga de trabalho que faz escritas assíncronas até atingir o tamanho limite de mudança de log e que, posteriormente, faz algumas escritas assíncronas, seguidas de uma síncrona, no novo log. De seguida, montou-se o LazyFS+ na diretoria onde os dados do LevelDB são guardados e aplicou-se a carga de trabalho. Após a finalização da mesma, solicitou-se a limpeza da cache através do comando `lazyfs::clear-cache`. Após receber a confirmação de que a falta foi injetada, tentou-se ler os valores presentes na base de dados do LevelDB.

Observações e causa. O utilizador que descreveu este problema indica que ocorreu uma perda parcial dos dados na versão 1.12 do LevelDB. Esta foi a primeira versão testada com o LazyFS+. Constatou-se, no entanto, que após aplicar os passos descritos no parágrafo anterior, o LevelDB v1.12 reporta o seguinte erro "*Corruption: CURRENT file does not end with newline*", em vez de apresentar a corrupção indicada pelo utilizador. Esta mensagem de erro induz a pensar que algum conteúdo do ficheiro `CURRENT` não foi persistido. No entanto, ao solicitar um pedido ao LazyFS+ para este mostrar um relatório dos dados não sincronizados, após a carga de trabalho ter sido aplicada, obtemos a informação apresentada no Listing 5.1.

```
[2023-10-29 15:35:36.237] [global] [info] [lazyfs.cmds]: report: [inode
   3671901] is not fully synced, info:
[2023-10-29 15:35:36.237] [global] [info] [lazyfs.cmds]: report: [inode
   3671901] info: (block 0) to (block 15) [byte index 0 to index 65535]
[2023-10-29 15:35:36.237] [global] [info] [lazyfs.cmds]: report: [inode
   3671901] files mapped to this inode:
[2023-10-29 15:35:36.237] [global] [info] [lazyfs.cmds]: report: [inode
   3671901] => file: '/home/gsd/leveldb1_12_test/test-r/000005.sst'
[2023-10-29 15:35:36.237] [global] [info] [lazyfs.cmds]: report: total
   number of bytes un-fsynced: 65536 bytes.


[2023-10-29 15:35:36.237] [global] [info] [lazyfs.cmds]: report: [inode
   3671903] is not fully synced, info:
[2023-10-29 15:35:36.238] [global] [info] [lazyfs.cmds]: report: [inode
   3671903] info: (block 0) to (block 1007) [byte index 0 to index 4128767]
[2023-10-29 15:35:36.238] [global] [info] [lazyfs.cmds]: report: [inode
   3671903] files mapped to this inode:
[2023-10-29 15:35:36.238] [global] [info] [lazyfs.cmds]: report: [inode
   3671903] => file: '/home/gsd/leveldb1_12_test/test-r/000003.log'
[2023-10-29 15:35:36.238] [global] [info] [lazyfs.cmds]: report: total
   number of bytes un-fsynced: 4194304 bytes.
```

```
[2023-10-29 15:35:36.238] [global] [info] [lazyfs.cmds]: report: [inode
   3671898] is not fully synced, info:
[2023-10-29 15:35:36.238] [global] [info] [lazyfs.cmds]: report: [inode
   3671898] info: (block 0) to (block 0) [byte index 0 to index 122]
[2023-10-29 15:35:36.238] [global] [info] [lazyfs.cmds]: report: [inode
   3671898] files mapped to this inode:
[2023-10-29 15:35:36.238] [global] [info] [lazyfs.cmds]: report: [inode
   3671898] => file: '/home/gsd/leveldb1_12_test/test-r/LOG'
[2023-10-29 15:35:36.238] [global] [info] [lazyfs.cmds]: report: total
   number of bytes un-fsynced: 4194427 bytes.
```

Listing 5.1: Relatório do LazyFS+ relativo aos dados em cache em risco de serem perdidos do LevelDB v1.12.

É possível verificar que os ficheiros com dados não sincronizados são o `LOG`, `00003.log` e `000005.sst`. O ficheiro com a extensão `log` corresponde ao log antigo e o ficheiro com a extensão `sst` corresponde à tabela ordenada criada a partir desse log. Apesar do LevelDB indicar um problema ao nível do ficheiro `CURRENT`, nenhum conteúdo deste ficheiro se encontra em risco de ser perdido. Com o intuito de confirmar que a causa do problema na leitura da base de dados era efetivamente a não persistência dos dados em cache, realizou-se um segundo teste em que se solicitou ao LazyFS+ que persistisse todos os dados em cache. Quando todos os dados são persistidos, nenhuma mensagem de erro é reportada e todos os pares chave-valor se encontram na base de dados.

Conseguimos encontrar um outro utilizador que se deparou exatamente com o mesmo erro [1]. Em comentários ao relatório do utilizador, é indicado que o *bug* foi corrigido.

Como o resultado na versão 1.12 não correspondeu ao resultado reportado pelo utilizador, tentamos reproduzir o problema na versão mais recente do LevelDB, a versão 1.23. Nesta versão, obteve-se o resultado mencionado: os pares chave-valor que foram escritos no log que foi compactado para uma tabela foram todos perdidos, enquanto que os pares que foram escritos de forma assíncrona no novo ficheiro, devido ao facto de serem seguidos de uma escrita síncrona, foram persistidos.

Possível resolução. A nossa sugestão para evitar este problema, que foi confirmada com o LazyFS+, passa por fazer `fsync` dos ficheiros `sst` e log, que contêm as escritas assíncronas que foram perdidas.

Perda de metadados no LevelDB e no PebblesDB Ⓑ

Um utilizador do LevelDB reportou que a sua base de dados ficou corrompida, sendo que é fornecida a seguinte mensagem de erro "*Corruption: no meta-nextfile entry in descriptor*"[2]. O relatório deste *bug*

[1] *"LevelDBError: Corruption: CURRENT file does not end with newline"database recovery was not possible* - https://github.com/google/leveldb/issues/75

[2] *feature request: repair leveldb databases* - https://github.com/omniscale/imposm3/issues/118

carece de informações detalhadas sobre a forma como o problema foi causado. No entanto, é possível que resulte de escritas incompletas nos ficheiros de metadados do LevelDB, pois a mensagem de erro indica a falta de informação ao nível dos metadados.

Como foi mencionado na Secção 3.1, o ficheiro `MANIFEST` do LevelDB é responsável por guardar metadados. Neste sentido, este foi o ficheiro alvo da falta injetada. O mesmo teste foi realizado no PebblesDB, já que esta aplicação tem como base de implementação o LevelDB.

Estratégia de injeção. Depois de iniciar o LazyFS+, injetou-te uma falta de limpeza da cache para ocorrer imediatamente após a escrita para o ficheiro `MANIFEST`. Posteriormente, executou-se uma carga de trabalho com a escrita de alguns pares chave-valor.

Observações e causa. Após a falta programada ser injetada, o LevelDB e o PebblesDB falham ao reiniciar, com a mensagem de erro que foi indicada pelo utilizador. Uma das informações que o ficheiro de metadados contém é o número do próximo ficheiro `sst`. Como o ficheiro `MANIFEST` não contém essa informação, já que a escrita não foi persistida, o erro é reportado.

Possível resolução. Neste caso, é sugerido executar a função `RepairDB` do LevelDB, que deve ser utilizada em casos de corrupção. Confirmamos que a sua utilização corrige com sucesso o erro.

Ficheiro de log vazio no Zookeeper Ⓒ

Este *bug* [3] indica que o ZooKeeper falha ao reiniciar devido a um ficheiro de log vazio e foi detetado, pela primeira vez, na versão 3.4.6 do ZooKeeper.

A aplicação foi testada na versão 3.4.8, já que esta é uma versão onde existem outros *bugs* que foram reproduzidos neste estudo. Adicionalmente, foi verificado numa das versões mais recentes e ainda ativamente suportada pelo ZooKeeper, a versão 3.7.1, se era possível encontrar o problema, já que este se encontrado assinalado como por resolver. Confirmou-se, com o LazyFS+, que este é um *bug* ainda existente atualmente.

Estratégia de injeção. Para reproduzir esta falta foi criado um *cluster* com 3 nodos. Após iniciar o LazyFS+ e o ZooKeeper, foi programada uma falta a ser injetada antes da próxima operação de `fsync` do ficheiro de log. Posteriormente, estabeleceu-se uma conexão entre um cliente e o servidor onde a falta foi injetada.

[3] *Zookeeper failed to start for empty txn log* - https://issues.apache.org/jira/browse/ZOOKEEPER-2332?jql=project%20%3D%20ZOOKEEPER%20AND%20resolution%20%3D%20Unresolved%20AND%20text%20~%20%22empty%20snapshot%22%20ORDER%20BY%20priority%20DESC%2C%20updated%20DESC

Observações e causa. Após a falta ser injetada e o LazyFS+ ser montado novamente, tentou-se iniciar o servidor do ZooKeeper alvo da falta. Este falhou em reiniciar. Na versão 3.4.8, a mensagem de erro imprimida é "*Unable to load database on disk*", já na versão 3.7.1 o erro é "*No snapshot found, but there are log entries*". Apesar de na segunda mensagem de erro aparentemente termos mais informação sobre o problema que ocorreu, este é enganador, já que, na realidade, existem dois ficheiros de *snapshot*, um com 0 *bytes* e outro com 604 *bytes* e um ficheiro de log vazio. Isto é possível verificar, quer através da análise da diretoria de ficheiros, quer através do log do LazyFS+.

O servidor do ZooKeeper falhou depois de ter sido criado o ficheiro de log, mas antes de o seu cabeçalho ter sido escrito para disco. Consequentemente, o ficheiro de log ficou sem um cabeçalho válido, o que subsequentemente levou à falha do servidor na sua reinicialização. A origem deste problema consiste na existência deste ficheiro de log inválido. De modo a confirmar esta hipótese, antes de reiniciar o servidor do ZooKeeper, experimentou-se remover manualmente o ficheiro de log, o que resultou, posteriormente, numa reinicialização com sucesso do servidor.

Possível resolução. Um dos comentários no relatório reportado pelo utilizador aponta para a necessidade de existirem garantias de atomicidade na atualização deste log e sugere a seguinte sequência de passos para resolver este problema:

1. Criar o ficheiro `log.<zxid>.new` (o nome dos logs do ZooKeeper são sempre `log.<zxid>`, sendo *zxid* um identificador de transação, utilizado para expor ordem total).
2. Escrever o cabeçalho e persistir o ficheiro.
3. Fazer um `rename` atómico de `log.<zxid>.new` para `log.<zxid>`.

Esta estratégia é tipicamente utilizada para evitar estados intermédios na escrita de ficheiros.

Ficheiro de WAL vazio no etcd Ⓓ

Tendo em conta o problema Ⓒ do ZooKeeper, verificou-se se o mesmo não ocorreria no ectd. Na versão 2.3.0 do etcd, testámos a capacidade do sistema de tolerar a ocorrência de uma falha na primeira escrita realizada para o ficheiro de WAL. Optamos por testar essa versão mais antiga devido à reprodução de outros *bugs* encontrados nessa versão, os quais serão mencionados posteriormente.

Estratégia de injeção. Este *bug* foi encontrado ao correr uma instância do etcd. Depois de iniciar o LazyFS+, é necessário programar uma falta a ser injetada após a próxima escrita no ficheiro de WAL (como o etcd só é iniciado após recebermos a confirmação de que o LazyFS+ recebeu esta falta, significa que

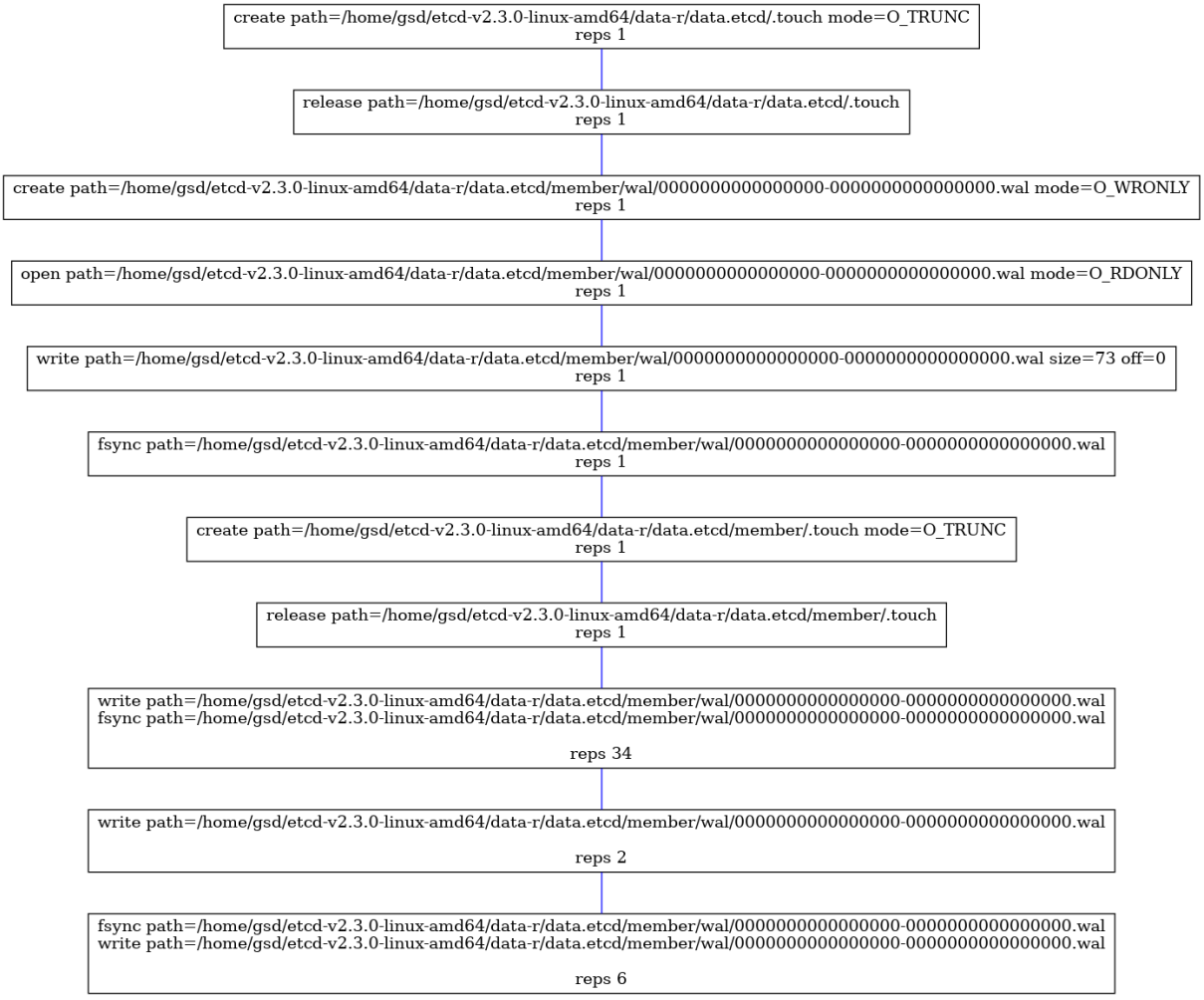


Figura 7: Sequência de chamadas ao sistema do v2.3.0 etcd no momento de inicialização, onde nenhuma falta foi aplicada. Imagem produzida pelo *parser* do LazyFS+.

esta é injetada após a primeira escrita e antes de qualquer `fsync`). Depois de receber a confirmação de que a falta foi processada pelo LazyFS+, iniciar o etcd com a diretoria de dados sobre o LazyFS+. Passado pouco tempo, o etcd e o LazyFS+ irão falhar, já que a falta será injetada no arranque do etcd. Como é possível ver na Figura 7, o ficheiro alvo da falta é criado e escrito logo no início da execução.

Observações e causa. Após a injeção da falta, o etcd falha a reiniciar com a mensagem de erro "*etcdserver: read wal error (wal: snapshot not found) and cannot be repaired*". Tal como no caso do ZooKeeper, a mensagem de erro reportada é enganadora, pois o problema não é o facto de não existir *snapshot*, mas sim existir um ficheiro de WAL vazio. Efetivamente, o ficheiro de *snapshot* não é criado, no entanto, caso uma falha semelhante seja injetada já depois de algum conteúdo do ficheiro de WAL ter sido persistido, nenhum erro é reportado, mesmo que não exista qualquer ficheiro de *snapshot*. A causa deste problema é semelhante ao erro do ZooKeeper: a primeira escrita para o ficheiro de WAL corresponde ao seu cabeçalho e o etcd necessita de um cabeçalho válido para ser capaz de reiniciar.

Possível resolução. Numa das versões mais atuais do etcd (v3.4.25), este problema já não acontece. Na Figura 8 é possível ver a sequência de chamadas ao sistema executadas pelo etcd v3.4.25. Verifica-se que, em vez de se escrever diretamente no ficheiro de WAL, o etcd escreve para um ficheiro numa diretoria temporária (`wal.tmp`), faz `fsync` do ficheiro de WAL nessa diretoria e, posteriormente, renomeia a diretoria de `wal.tmp` para `wal`.

Inconsistência de dados no ZooKeeper Ⓔ

O segundo *bug* identificado na versão 3.4.8 do ZooKeeper é notavelmente mais complexo de reproduzir e acarreta consequências mais graves. Este problema surgiu durante a tentativa de reproduzir o *bug* Ⓒ e tem como consequência a inconsistência de dados num *cluster* do ZooKeeper. Foi verificado, com o LazyFS+, se o *bug* ocorria numa das versões mais recentes e confirmou-se que este foi corrigido.

Uma data de problemas reportados relatam inconsistências de dados a ocorrer no ZooKeeper [90, 109, 110, 123], no entanto, as informações disponíveis não são suficientemente claras para estabelecer uma conexão definitiva entre a causa deste *bug* com o relatado nesses problemas.

Estratégia de injeção. Para reproduzir esta falta foi utilizado um *cluster* de 3 nodos, sendo que apenas num dos nodos foi usado o LazyFS+ como SF. Vamos considerar que os nodos são identificados por *nodo1*, *nodo2* e *nodo3* e que no *nodo2* o ZooKeeper possui os seus ficheiros numa diretoria do LazyFS+. Os seguintes passos foram executados:

1. Após iniciar o ZooKeeper em todos os nodos, conectar um cliente ao *nodo1*.

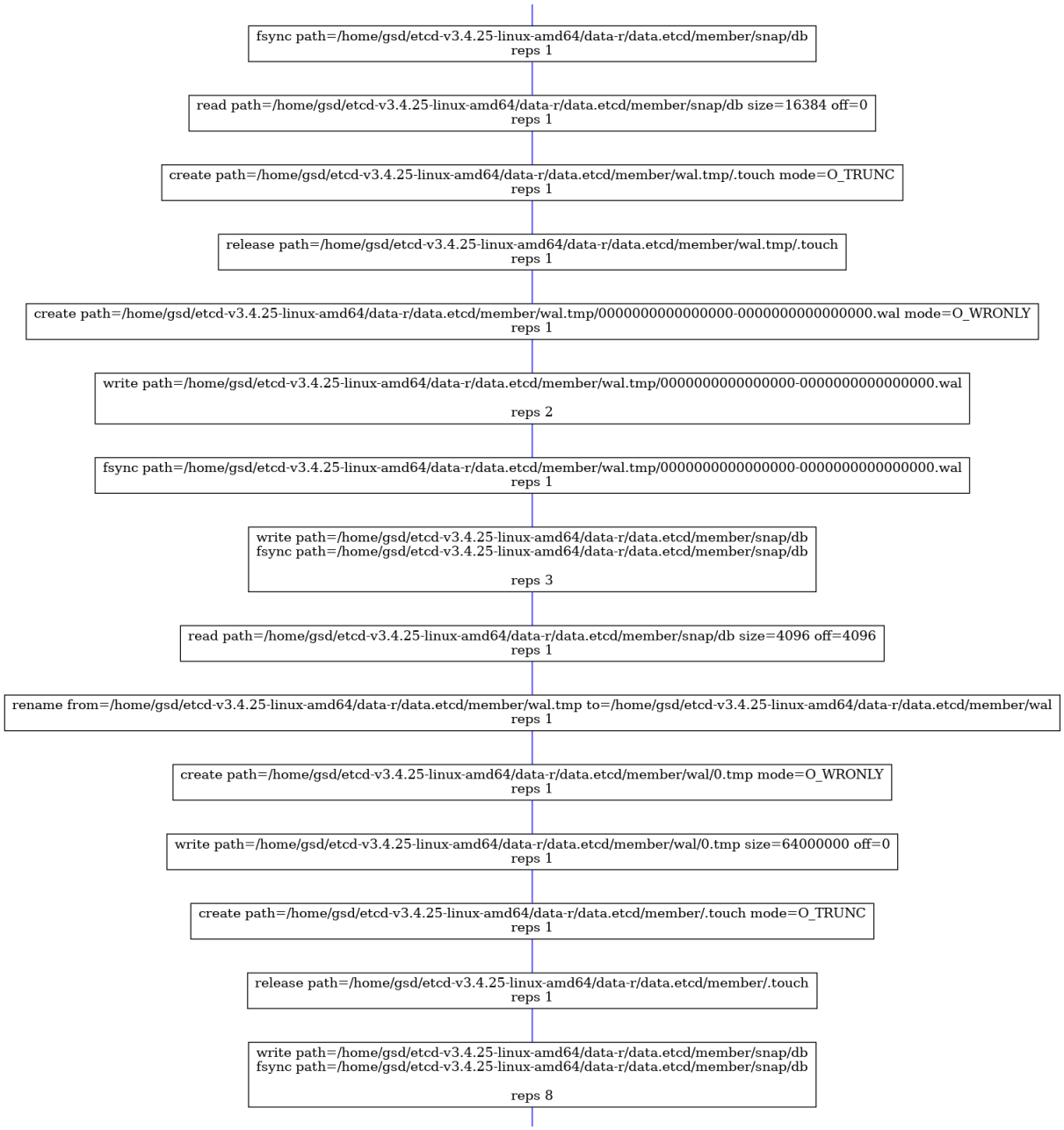


Figura 8: Sequência de chamadas ao sistema executadas pelo etcd v3.4.25 no momento de inicialização, sem que nenhuma falta seja aplicada. Imagem produzida pelo *parser* do LazyFS+.

2. Criar o *znode* `/a` com uma *string* com poucos caracteres como os dados desse *znode*. Esta operação traduz-se na execução do comando `create /a my_data`.

3. Injetar uma falta de falha para acontecer antes do próximo `write` no log do *nodo2*.

4. Após o LazyFS+ imprimir no log que a falta foi programada com sucesso, criar o *znode* `/b`. Esta operação terá como consequência a falha do *nodo2*.

5. Criar o *znode* `/c` no *nodo1*, desconectar o cliente e reiniciar o *nodo2*.

6. Conectar o cliente ao *nodo2* e verificar que os três *znodes* previamente inseridos estão presentes. Injetar a mesma falta do ponto *3*.

7. Com o cliente conectado ao *nodo2* e após receber a confirmação de que a falta foi programada, criar o *znode* `/d`. Novamente, o *nodo2* irá falhar.

8. Reiniciar o *nodo2* e verificar que este nodo apenas possui os *znodes* `/a` e `/d`, ao passo que os restantes servidores possuem todos os *znodes* criados: `/a`, `/b`, `/c` e `/d`.

Observações e causa. No Listing 5.2 é possível verificar que no *nodo2*, o *znode* `/b` não existe, mas ao tentar criá-lo, o cliente é informado do contrário, porque está presente noutros servidores.

```
[zk: 192.168.112.88:2181(CONNECTED) 0] create /b aba
Node already exists: /b
[zk: 192.168.112.88:2181(CONNECTED) 1] get /b
Node does not exist: /b
```

Listing 5.2: Interação de um cliente com o servidor do ZooKeeper onde foram injetadas as faltas.

Se o valor do *znode* `/b` fosse alterado no *nodo2*, o cliente nunca recebia resposta, os outros nodos eram atualizados, e o servidor do *nodo2* reportava o erro apresentado no Listing 5.3. O mesmo ocorre para o *znode* `/c`.

```
2023-02-03 17:22:14,866 [myid:1] - WARN [CommitProcessor:1
:NIOServerCnxn@1097] - Unexpected exception. Destruction averted.
java.lang.NullPointerException
at org.apache.jute.BinaryOutputArchive.writeRecord(BinaryOutputArchive.java
    :123)
at org.apache.zookeeper.proto.SetDataResponse.serialize(SetDataResponse.
    java:39)
at org.apache.jute.BinaryOutputArchive.writeRecord(BinaryOutputArchive.java
    :123)
at org.apache.zookeeper.server.NIOServerCnxn.sendResponse(NIOServerCnxn.
    java:1073)
```

```
at org.apache.zookeeper.server.FinalRequestProcessor.processRequest(
    FinalRequestProcessor.java:404)
at org.apache.zookeeper.server.quorum.CommitProcessor.run(CommitProcessor.
    java:77)
```

Listing 5.3: Interação de um cliente com o servidor do ZooKeeper onde foram injetadas as faltas. Resposta dada pelo servidor quando o cliente realiza um `set` para os *znodes* `/b` e `/c`.

Resumidamente, o nodo onde as faltas foram injetadas tem uma visão diferente dos dados do que os restantes nodos do *cluster*. Enquanto que os outros nodos conhecem os *znodes* `/a`, `/b`, `/c` e `/d`, o *nodo2* apenas conhece `/a` e `/d`, apesar de que, após a primeira falta, conhecia também os *znodes* `/b` e `/c`.

Este problema de inconsistência resulta da não sincronização do ficheiro de *snapshot*. Após as faltas serem injetadas, o LazyFS+ reporta que existem blocos de dados do ficheiro de *snapshot* que não tinham sido persistidos 5.4.

```
[2023-09-05 15:40:24.889] [global] [info] [lazyfs.cmds]: report: [inode
    3802475] is not fully synced, info:
[2023-09-05 15:40:24.889] [global] [info] [lazyfs.cmds]: report: [inode
    3802475] info: (block 0) to (block 0) [byte index 0 to index 295]
[2023-09-05 15:40:24.889] [global] [info] [lazyfs.cmds]: report: [inode
    3802475] files mapped to this inode:
[2023-09-05 15:40:24.889] [global] [info] [lazyfs.cmds]: report: [inode
    3802475] => file: '/home/gsd/data/zookeeper-root/version-2/snapshot
    .100000000'
[2023-09-05 15:40:24.889] [global] [info] [lazyfs.cmds]: report: total
    number of bytes un-fsynced: 296 bytes.
```

Listing 5.4: Relatório do LazyFS+ do ZooKeeper v3.4.8 na injeção das faltas levam à inconsistência.

Possível resolução. Tendo em conta que o ficheiro de *snapshot* não é sincronizado, repetimos o mesmo teste, sendo que solicitamos ao LazyFS+ que este realizasse uma sincronização de todos os dados em cache, utilizando o comando `lazyfs::cache-checkpoint`, antes de as duas faltas mencionadas serem injetadas. Como resultado, no fim da experiência, nenhuma inconsistência surgiu.

Tentamos reproduzir este *bug* numa das versões mais recentes do ZooKeeper, no entanto, esta inconsistência já não acontece na versão 3.7.1. Verificou-se que, após a injeção de cada falta, o LazyFS+ deixa de reportar a existência de blocos de dados do ficheiro de *snapshot* por persistir.

5.2.2 Faltas de reordenação

A reordenação de escritas nas aplicações testadas teve como principais consequências a indisponibilidade e a perda silenciosa de dados. Para cada aplicação onde foram encontrados problemas, será feita uma

descrição de cada *bug*.

Incompatibilidade do número mágico no ZooKeeper Ⓕ

Com a reprodução do *bug* Ⓒ , que ocorreu no ZooKeeper, foi possível estudar parte do fluxo de operações realizado por esta aplicação. Verificou-se que, quando um cliente se conecta a um servidor, são realizadas várias escritas consecutivas para o ficheiro de log. A Figura 9 mostra parte do grafo produzido pelo LazyFS+ onde é explícita a chamada consecutiva de `writes`. Como estas escritas podem constituir uma vulnerabilidade, pois podem ser reordenadas, experimentou-se na versão 3.4.8 e numa versão mais recente, a 3.7.1, injetar a falta de reordenação.

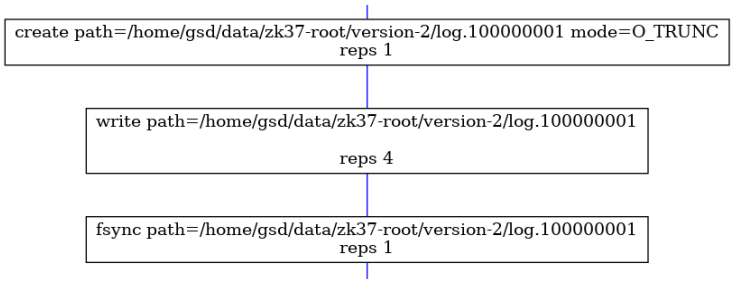


Figura 9: Sequência de chamadas ao sistema executadas pelo ZooKeeper v3.7.1 quando o cliente se conecta a um servidor, sem que nenhuma falta tenha sido injetada. Imagem produzida pelo *parser* do LazyFS+.

Estratégia de injeção. Para esta falta, apenas foi necessário iniciar um *cluster* composto por 3 nodos e usar LazyFS+ como SF num dos nodos. No ficheiro de configuração do LazyFS+ foi adicionada uma falta de reordenação para o primeiro grupo de escritas consecutivas para o ficheiro de log, sendo que apenas o terceiro `write()` será persistido. Após o *cluster* estar disponível, conecta-se o cliente ao servidor com o LazyFS+.

Observações e causa. Após a injeção de falta, tentou-se reiniciar o servidor onde esta foi injetada. O servidor falha ao iniciar com o erro"*Transaction log has invalid magic number 0 != 1514884167*". Quando o cliente se conecta, são realizados quatro `writes` consecutivos para o ficheiro de log. O primeiro `write()` é responsável pela escrita de um número mágico. Os números mágicos são utilizados, por exemplo, para verificar o formato do ficheiro, compatibilidade de versões, integridade de dados, segurança, entre outros. Ao reiniciar o servidor, o log irá conter dados resultantes da persistência da terceira escrita, mas o número mágico não irá corresponder àquele que está escrito no ficheiro de log, porque a primeira escrita não foi persistida.

Após obter o erro reportado pelo ZooKeeper, foi possível encontrar *online* um utilizador que se deparou com este problema [4]. Este não foi capaz de fornecer informações acerca da potencial causa do *bug* ou se quer de como reproduzi-lo, o que não é surpreendente, pois, muitas vezes, os utilizadores não têm acesso a ferramentas como o LazyFS+ para perceber as causas deste tipo de corrupções.

Possível resolução. Uma possível resolução é garantir a persistência do número mágico (críticos para o sistema) e que, neste caso, se traduz na persistência da primeira escrita. É possível, com o LazyFS+, confirmar esta solução, pois basta configurar uma falta de reordenação em que apenas o primeiro `write` é persistido. Neste cenário de injeção, o servidor, após a falta, conseguiu reiniciar com sucesso.

Incompatibilidade de *checksums* no ZooKeeper Ⓖ

O segundo *bug* encontrado no ZooKeeper foi aquele que inspirou a introdução das faltas de reordenação no LazyFS+. Este resulta na indisponibilidade do servidor do ZooKeeper onde ocorre a falha devido a uma incompatibilidade de *checksums*.

O *bug* foi encontrado pelo PACE, sendo que o autor do artigo referente a esta ferramenta também reportou o problema *online* [5]. Este foi originalmente encontrado na versão 3.4.8 e, com o LazyFS+, foi possível confirmar que o *bug* ainda ocorre numa das versões recentes, a versão 3.7.1.

Estratégia de injeção. Para reproduzir este *bug* é necessário iniciar um *cluster* com 3 nodos, sendo que apenas em deles utiliza o LazyFS+ como SF na diretoria de dados do ZooKeeper. O LazyFS+ deve ser configurado para injetar uma falta de reordenação no ficheiro de log, em que algumas escritas são persistidas (a 1ª, 4ª, 7ª e 8ª, mais concretamente), pois, como é possível ver na Figura 10, várias escritas consecutivas são realizadas. Após os servidores estarem iniciados, executa-se uma carga de trabalho com várias escritas.

Observações e causa. Após a falta ser injetada e o LazyFS+ ser montado novamente, tentou-se iniciar o servidor do ZooKeeper alvo da falta. Este falhou em reiniciar como erro "*CRC check failed*", como era esperado.

O ZooKeeper é capaz de identificar com sucesso que o ficheiro de log foi incorretamente atualizado, pois utiliza *checksums* para a verificação da sua integridade. Já que existem outros nodos no *cluster*, o nodo onde a falta foi injetada poderia recorrer a outros nodos para corrigir a corrupção detetada, no entanto, não o faz. Para além disso, existe, na versão 3.7.1, um *script* do próprio ZooKeeper que permite

[4] *Transaction log: /logs/jhe/nc/apps/zookeeper/version-2/log.1400470027 has invalid magic number 0 != 1514884167* - `https://lists.apache.org/thread/dlhrd5t2xyjr425l78xofqs7r3v4s486`

[5] *Possible Cluster Unavailability* - `https://issues.apache.org/jira/browse/ZOOKEEPER-2560`

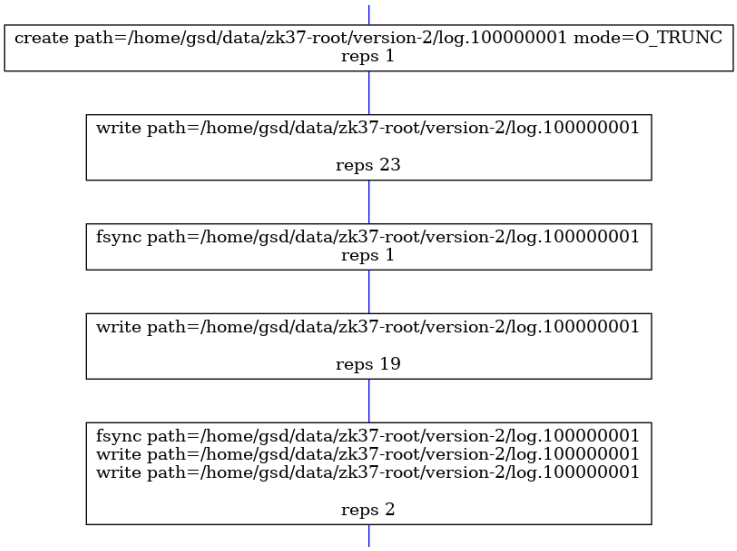


Figura 10: Parte da sequência de chamadas ao sistema realizadas pelo ZooKeeper v3.7.1 quando a carga de trabalho é executada, sem a injeção de faltas. Imagem produzida pelo *parser* do LazyFS+.

recuperar entradas do log que estão corrompidas [76]. No entanto, quando se tentou correr este *script*, obteve-se o erro "*Unsupported Txn with type=%d0*".

Possível resolução. Uma solução simples (mas custosa) para este problema é fazer `fsync` após cada escrita. A falha do servidor ocorre porque a 2ª, 3ª, 5ª e 6ª escritas não foram persistidas. Simulou-se, com o LazyFS+, injetar uma falta após a 8ª escrita, sendo que todas as anteriores foram persistidas antes da falha. Neste cenário, em que todas as escritas foram persistidas e não ocorreu reordenação, o ZooKeeper reinicia com sucesso.

Incompatibilidade de *checksums* no etcd Ⓗ

Este *bug* foi encontrado na versão 2.3.0 do etcd pelo PACE [44]. Na Figura 8 é possível constatar que existem duas escritas consecutivas. A reordenação destas leva à falha do etcd.

Estratégia de injeção. No ficheiro de configuração do LazyFS+ foi programada uma falta de reordenação para o primeiro grupo de escritas consecutivas para o ficheiro de WAL, sendo que só a segunda escrita é persistida. Não foi necessário executar qualquer carga de trabalho, porque nesta versão do etcd, são realizadas escritas periodicamente para o ficheiro de WAL.

Observações e causa. O etcd falha ao reiniciar reportando a mensagem de erro "*(walpb: crc mismatch) and cannot be repaired*". Tal como no caso anterior, a aplicação é capaz de detetar a ocorrência de problemas, pois utiliza *checksums*, porém, não possui mecanismos automáticos para recuperar destes problemas.

Possível resolução. Uma solução para este problema resume-se a fazer `fsync` após a primeira escrita. De modo a validar esta solução, foi simulada uma falha após persistir a primeira escrita e verificou-se que, neste cenário, o etcd consegue corrigir o problema com o ficheiro de WAL. No log do etcd é reportado "`wal: repairing data/data.etcd/member/wal/0000000000000000-0000000000000000.wal`". O mesmo se aplica quando já existem pares chave-valor guardados, sendo que o etcd é capaz de corrigir o ficheiro de WAL sem os descartar.

Corrupção de dados no LevelDB ①

Este *bug* foi mencionado na Secção 3.3 e é um dos problemas encontrados que traz consequências mais graves para os utilizadores. Resumidamente, verificou-se que após a ocorrência de uma falha de energia, alguns valores de uma base de dados do LevelDB ficaram corrompidos. A execução da função recomendada para a correção da corrupção leva a que, em vez dos valores corrompidos serem descartados ou completamente recuperados, estes sejam parcialmente recuperados.

A estratégia de reprodução do *bug* sugerida pelo utilizador que o encontrou inclui alterar o código fonte do LevelDB e cortar de forma abrupta a fonte de energia do servidor.

Fase de preparação. Para reproduzir este problema com o LazyFS+, foi necessário perceber os pontos em que poderiam ser injetadas faltas. Deste modo, desenvolveu-se uma carga de trabalho que adiciona vários pares chave-valor, em que alguns dos valores têm um tamanho superior a 4 KiB. Depois analisou-se as chamadas ao sistema realizadas pela aplicação, executando a carga de trabalho e analisando o log produzido pelo LazyFS+. Visto que o *bug* tem como consequência a perda parcial de um valor, a causa do problema é, potencialmente, a reordenação e divisão de escritas.

Na Figura 11 é possível identificar um conjunto de escritas que são realizadas consecutivamente. Experimentou-se, portanto, reordenar esse conjunto de escritas e conseguiu-se, com sucesso, reproduzir o *bug*.

Estratégia de injeção. O LazyFS+ foi configurado para persistir a 1ª, 3ª e 5ª escritas, do primeiro grupo de escritas consecutivas, no ficheiro `000003.log`. De seguida, o LazyFS+ foi montado na diretoria de dados do LevelDB e executou-se a carga de trabalho. De realçar que o LevelDB foi configurado para reportar erro em caso de corrupção. Para corrigir os dados corrigidos, aplicou-se a função recomendada.

Observações e causa. A escrita de um valor com tamanho superior a uma página foi divida em várias escritas. A reordenação destas escritas levou a que o valor ficasse incompleto. A execução da função `RepairDB` levou a que a parte do valor em falta (que estava preenchida com o carater `NULL`) fosse eliminada e, consequentemente, o valor fosse restaurado incorretamente.

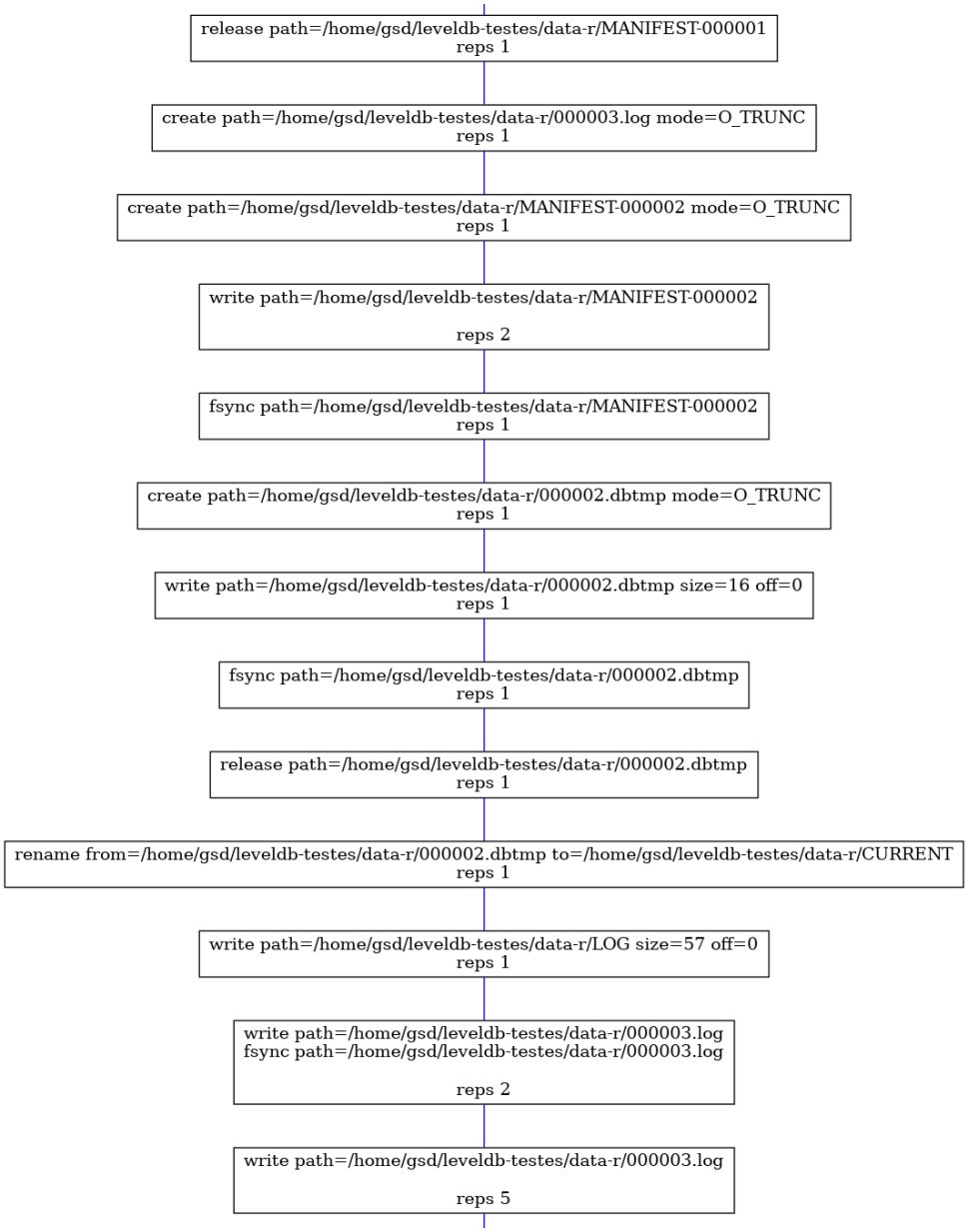


Figura 11: Conjunto de chamadas ao sistema invocadas na execução de uma carga de trabalho no LevelDB 1.15, sem a injeção de faltas. Imagem produzida pelo *parser* do LazyFS+.

No caso em que não é acionada a opção de o LevelDB reportar corrupção, este simplesmente descarta o par chave-valor corrompido.

Possível resolução. Tal como nos casos anteriores, uma possibilidade é tornar esta escrita atómica recorrendo a um ficheiro temporário e ao `rename`.

Este problema ainda se encontra assinalado como por resolver no repositório do LevelDB e alguns utilizadores queixam-se de se terem deparado com ele em versões posteriores. Com o LazyFS+ é possível verificar rapidamente que na versão mais recente, a versão 1.23, o problema já não ocorre. Constatou-se que, apesar de existir um grupo de duas escritas consecutivas e uma escrita de tamanho superior a 4 , nenhum problema ocorre quando se injetam faltas de reordenação e atomicidade. Qualquer escrita que potencialmente tenha sido corrompida é descartada.

5.2.3 Faltas de atomicidade

Os seguintes *bugs* resultam a da injeção de faltas de atomicidade, isto é, faltas que dividem escritas em escritas menores.

Incompatibilidade de *checksums* no etcd Ⓙ

Este *bug* foi encontrado na versão 2.3.2 por um utilizador do etcd [6] e foi reproduzido no LazyFS+ com a versão 2.3.0. O erro fornecido pelo etcd indica uma incompatibilidade de *checksums*. No relatório fornecido pelo utilizador, é indicada a possibilidade do problema resultar de um *torn write*, isto é, da divisão da escrita de um conjunto de dados em várias escritas de porções desse conjunto. Outro problema reportado com o mesmo erro [7], numa versão mais recente do etcd, não foi considerado pela equipa da aplicação devido à falta de meios para reproduzir [8].

Fase de preparação. Dada a falta de informação de como reproduzir este *bug*, foi necessário analisar o fluxo de operações executadas pelo etcd. Como a causa deste problema são escritas grandes, foi desenvolvida uma carga de trabalho com a inserção de vários valores, sendo que alguns destes possuem acima de 4 (o tamanho de uma página da cache). Após executar esta carga de trabalho com o LazyFS+ como SF na diretoria de dados do etcd, utilizou-se o *parser* do log para localizar escritas maiores que uma página. Obteve-se a informação de que a 35ª escrita para o ficheiro de WAL tem tamanho superior ao de uma página.

[6] *ETCD data gets corrupted with error "read wal error (walpb: crc mismatch)"* - https://github.com/etcd-io/etcd/issues/6191#issuecomment-240268979

[7] *ETCD data gets corrupted with error "walpb: crc mismatch"* - https://github.com/etcd-io/etcd/issues/11488

[8] https://github.com/etcd-io/etcd/issues/11488#issuecomment-570241825

Estratégia de injeção. Este *bug* foi encontrado ao correr uma instância do etcd. No ficheiro de configuração do LazyFS+ foi programada uma falta de atomicidade em que a 35ª escrita ficheiro de WAL é dividida em duas escritas. Foram realizados dois testes distintos: um em que a primeira escrita é persistida e outro onde é a segunda. Depois de iniciar o LazyFS+ e o etcd, executa-se a carga de trabalho desenvolvida. Nos dois casos, obteve-se o mesmo resultado.

Observações e causa. O resultado é o mesmo que o do *bug* causado pela reordenação de escritas Ⓗ . Neste caso, verificou-se que, independentemente da porção da escrita persistida, o etcd falha, o que indica a necessidade da escrita ser atómica e que dividir a escrita em duas e persistir separadamente cada uma dessas escritas não é uma solução possível.

Possível resolução. Uma possível solução para este problema passa por realizar esta escrita crítica para um ficheiro temporário, fazer `fsync` do mesmo e fazer um `rename` do ficheiro temporário para o nome final do ficheiro.

Falha no arranque do etcd Ⓚ

Visto que foram encontrados *bugs* na versão 2.3.0 do etcd, verificou-se se existiam vulnerabilidades numa versão mais recente do etcd, a versão 3.4.25. Ao iniciar o etcd sem executar qualquer tipo de carga de trabalho, o *parser* do LazyFS+ identificou algumas escritas maiores que o tamanho típico de uma página, tal como é possível observar no Listing 5.5. Deste modo, injetamos faltas com o LazyFS+ nos diferentes ficheiros e verificamos a ocorrência de falhas no arranque do etcd quando se injetam faltas no ficheiro `member/snap/db`. Estas falhas foram reportadas no GitHub, com uma especificação de como utilizar o LazyFS+ para as reproduzir, e encontram-se atualmente em análise pelo etcd[9,10,11] .

```
> /home/gsd/etcd-v3.4.25-linux-amd64/data-r/data.etcd/member/snap/db
    - size: 16384 , off: 0 , occurrence: 1
    - size: 16785408 , off: 16384 , occurrence: 2

> /home/gsd/etcd-v3.4.25-linux-amd64/data-r/data.etcd/member/wal.tmp
   /0000000000000000-0000000000000000.wal
    - size: 64000000 , off: 0 , occurrence: 1

> /home/gsd/etcd-v3.4.25-linux-amd64/data-r/data.etcd/member/wal/0.tmp
    - size: 64000000 , off: 0 , occurrence: 1
```

Listing 5.5: Escritas com tamanho superior a 4 realizadas pelo etcd v3.4.25 e reportadas pelo LazyFS+.

[9] *etcd fails to start after power failure* - https://github.com/etcd-io/etcd/issues/16596

[10] *fault injection should honor page sizes* - https://github.com/dsrhaslab/lazyfs/issues/4

[11] *Update doc that db might be corrupted when power failure during initialization* - https://github.com/etcd-io/bbolt/issues/567

Estratégia de injeção. Este *bug* foi verificado tanto num *cluster* constituído por 3 nodos, como num composto por 1 nodo apenas. No primeiro caso, só num dos servidores é que o LazyFS+ foi montado na diretoria de dados do etcd. No ficheiro de configuração, foi programada uma falta de atomicidade para a primeira escrita no ficheiro `member/snap/db`. Foram realizadas três experiências distintas que levaram a erros:

1. a escrita é dividida em duas escritas de 8 e apenas a primeira é persistida;
2. a escrita é dividida em duas escritas de 8 e apenas a segunda é persistida;
3. a escrita é dividida em quatro escritas de 4 e apenas a primeira é persistida.

Nenhuma carga de trabalho foi aplicada nos testes realizados, ou seja, estes erros manifestam-se no arranque do sistema.

Observações e causa. As mensagens de erro reportadas pelo etcd nas diferentes situações foram as seguintes:

1. "*mvcc/backend: cannot open database at data/data.etcd/member/snap/db (invalid database)*"
2. "*fatal error: fault [signal SIGBUS: bus error code=0x2 addr=0x7f1a63138008 pc=0x8b3606]*"
3. "*mvcc/backend: cannot open database at data/data.etcd/member/snap/db (file size too small)*"

Estes *bugs* eram desconhecidos pelo etcd e a própria equipa do etcd reproduziu-os, com o LazyFS+, noutras versões, a 3.5 e 3.6. Aparentemente, estes problemas ocorrem devido ao facto de a escrita em questão ser de metadados do etcd. A equipa do etcd também verificou se o mesmo problema ocorria quando já existiam dados guardados e chegaram à conclusão que o problema só ocorre quando não existem dados. Apesar deste problema não colocar em causa os dados dos utilizadores, traz inconveniências, como afetar a redundância do *cluster*. Neste momento, estão a ser consideradas soluções.

Possível resolução. O terceiro teste foi realizado devido à proposta, por parte de um programador do etcd, de uma resolução para o problema [12]. Está sob consideração dividir a escrita crítica em quatro escritas, cada uma com tamanho de uma página e, após cada uma delas, ser realizado um `fsync`.

Esta estratégia, no entanto, não funciona, porque o que está em causa é a necessidade de a totalidade da escrita ser realizada ou não, ou seja, não podem existir porções desta escrita persistidas. Mesmo que não exista reordenação destas porções, que não ocorreria no caso de realizar um `fsync` a cada escrita, existem problemas. Isto foi confirmado com dois testes muitos simples:

[12] *avoid torn init writes* - https://github.com/etcd-io/bbolt/pull/570

- Dividiu-se a escrita de 16384 *bytes* em quatro escritas de tamanho semelhante, ou seja, de 4 , e persistiu-se apenas a primeira parte, de modo a simular uma falha de energia após a escrita e o `fsync` da mesma. Tal como foi dito anteriormente, este teste levou a um erro que indica que o tamanho do ficheiro é demasiado pequeno.

- Visto que a persistência da primeira parte da escrita crítica causa também erros, experimentou-se injetar uma falha no LazyFS+ antes da primeira escrita para o ficheiro `member/snap/db`, de modo a verificar se o etcd se queixa do facto do ficheiro estar vazio. Constatou-se que, quando o ficheiro tem 0 *bytes* de comprimento, a situação em que, após a criação do ficheiro, a energia falha, o etcd consegue reiniciar com sucesso.

Em sistemas Linux, é possível escrever atomicamente utilizando a estratégia, já mencionada, de escrita para ficheiro temporário, `fsync` do mesmo e `rename` desse ficheiro.

Corrupção de dados no PebblesDB Ⓛ

O PebblesDB, como já foi mencionado, tem como base de implementação uma versão do LevelDB com o problema da escrita parcial. Por conseguinte, testou-se o PebblesDB para verificar se o mesmo *bug* existia.

A estratégia para encontrar o *bug* foi semelhante à do LevelDB: aplicou-se a mesma carga de trabalho e analisou-se o fluxo de chamadas ao sistema executadas. No caso do LevelDB, existiam cinco escritas consecutivas para o mesmo ficheiro (Figura 11), mas o PebblesDB não faz o mesmo. Verificou-se, no entanto, que este realiza algumas escritas com tamanho superior a uma página, tal como é possível ver no Listing 5.6.

```
>  /home/gsd/pebblesDB-test/data-r/MANIFEST-000001
     - size:  262144 , off:  0 , occurrence:  1
>  /home/gsd/pebblesDB-test/data-r/MANIFEST-000002
     - size:  262144 , off:  0 , occurrence:  1
>  /home/gsd/pebblesDB-test/data-r/000003.log
     - size:  262144 , off:  0 , occurrence:  1
     - size:  12288 , off:  0 , occurrence:  4
```

Listing 5.6: Escritas de tamanho maior que uma página reportadas pelo *parser* do log do LazyFS+ no PebblesDB.

Procedeu-se à injeção de duas faltas distintas em que se dividiu as duas escritas para o ficheiro de log indicadas pelo *parser* do LazyFS+. A segunda escrita, de 12288 *bytes*, deu origem ao mesmo problema do LevelDB.

Estratégia de injeção. O LazyFS+ foi configurado para dividir a quarta escrita no ficheiro de log em 3 partes de tamanho semelhante e as 1ª e 3ª partes. De seguida, o LazyFS+ foi montado na diretoria de dados do PebblesDB e executou-se a carga de trabalho. De realçar que o PebblesDB foi configurado para reportar erro em caso de corrupção, tal como no caso do LevelDB, e foi utilizada a função recomendada para a correção deste problema (`RepairDB`).

Observações e causa. Tal como no caso do LevelDB, após a reparação da corrupção, o valor de uma determinada chave estava presente, mas incompleto, não correspondendo ao valor que foi inserido.

Possível resolução. Uma possível solução para evitar dados inválidos é descartar valores parciais, solução já adotada pela versão mais recente do LevelDB.

5.2.4 Validação e mecanismos de recuperação

O LazyFS+, para além de possibilitar a reprodução de *bugs*, permite também a validação do funcionamento dos mecanismos de recuperação das aplicações. Tanto o Redis como PostgreSQL aplicam estratégias que permitem recuperar de falhas de energia. Para ambos os casos, foram injetadas faltas de atomicidade e de reordenação de escritas.

Recuperação de dados corrompidos no Redis Ⓜ

De forma a conhecer o fluxo de operações executadas pelo Redis (versão 7.2.3), aplicou-se uma carga de trabalho muito simples, que consiste na inserção e atualização de uma chave. Na Figura 12 é possível ver as chamadas ao sistema resultantes desta execução.

Existem duas escritas que são realizadas consecutivamente, como é possível ver na Figura 12, e existem duas escritas realizadas para o ficheiro AOF que são de tamanho superior a uma página da cache. Procedeu-se, portanto, à injeção de uma falta de reordenação e a uma falta de atomicidade, separadamente.

Quando as faltas são injetadas, o Redis reporta imediatamente que ocorreu um erro na escrita do ficheiro e que, como está configurado para fazer `fsync` a cada escrita, é possível que o ficheiro de AOF não seja aceite quando o servidor for reiniciado. Após tentar reiniciar o servidor, dois cenários distintos surgiram:

- Nos dois tipos de faltas, no caso em que a primeira escrita é persistida, o Redis reinicia com sucesso e imprime a seguinte mensagem "*Short read while loading the AOF file appendonly.aof.1.incr.aof. Truncating the AOF appendonly.aof.1.incr.aof at offset 4141*". Ou seja, o Redis identifica a ocorrência de algum problema com o ficheiro e resolve-o imediatamente, eliminando os dados a partir

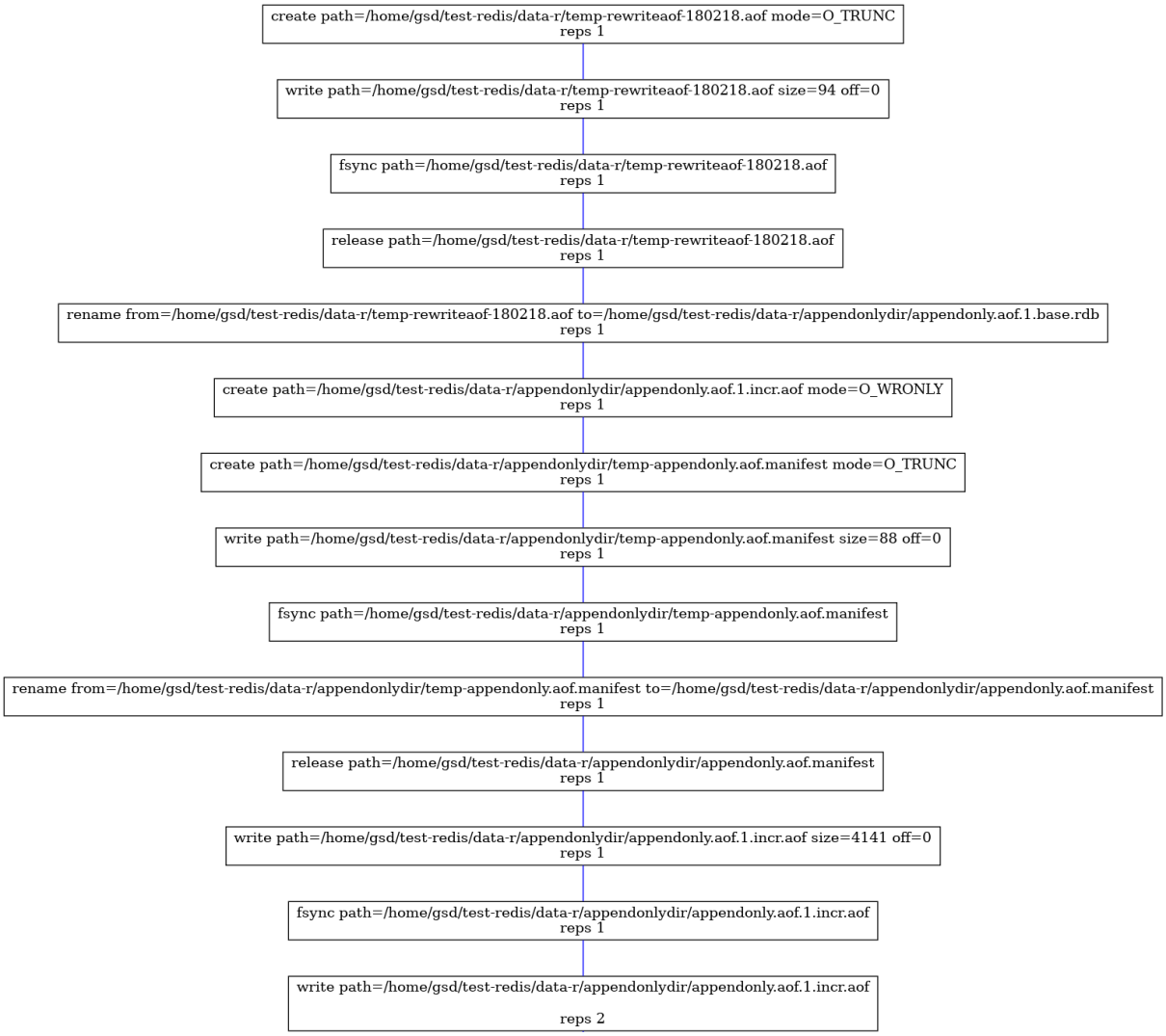


Figura 12: Chamadas ao sistemas executadas pelo Redis v7.2.3 quando é executada uma carga de trabalho, sem a injeção de faltas. Imagem produzida pelo *parser* do LazyFS+.

da posição 4141 do ficheiro AOF.

- Nos outros casos, em que a escrita persistida não é a primeira, o Redis falha ao iniciar e recomenda a utilização de um comando para reparar o ficheiro de AOF. Este comando identifica as posições válidas do ficheiro e pergunta ao utilizador se o pode truncar numa dada posição, que ele considere válido. Ao permiti-lo, quando é lido o valor da chave inserido, é lido o valor inicial, ou seja, como as faltas ocorreram na atualização do valor, essa operação foi descartada.

O primeiro caso descrito é o ideal, pois não é necessário a intervenção do utilizador para a reparação do servidor, no entanto, o segundo caso é uma solução melhor do que alguns casos analisados, em que o servidor falha e não existe forma de corrigir o problema.

Proteção contra escritas parciais no PostgreSQL Ⓝ

As tabelas e os índices do PostgreSQL são guardados em páginas de tamanho fixo, 8192 *bytes*, que devem ser escritos de forma atómica. O PostgreSQL utiliza um ficheiro de WAL para guardar todas as atualizações realizadas pelas transações em disco. Assim, a recuperação torna-se possível com a repetição das operações do ficheiro de WAL. Embora este ficheiro de log permita a recuperação de falhas comuns, não permite recuperar em casos de escritas parciais. A razão para isso relaciona-se com o facto de utilizar *logging* "físico-lógico", em que as entradas do log de transações indicam a página física a ser modificada e descrevem essa mudança em termos lógicos. Por exemplo, em vez de indicar a necessidade de reescrever dos *bytes* 10 a 20, este indica para eliminar o tuplo 7 e, para saber o que é este tuplo, é necessário consultar uma tabela de tuplos na página. Ou seja, para recuperar de corrupções do log, é necessário consultar o log [95].

Para remediar esta situação, o PostgreSQL periodicamente (nos chamados "*checkpoints*") escreve páginas inteiras para o ficheiro de WAL antes de modificar efetivamente a página em disco. A ideia é mudar para *logging* físico na primeira modificação realizada para uma página após um *checkpoint*, assim, em vez de ser indicado como modificar a página em termos lógicos, a totalidade da página é escrita no WAL.

Quando os dados são escritos, primeiro são persistido no ficheiro de WAL. Assim, se a energia falhar e causar uma escrita parcial da página, basta consultar o WAL. Se a energia falhar durante a escrita no WAL, como este possui *checksums* para cada entrada, a transação será descartada e a página permanecerá intacta.

O PostgreSQL permite desativar este comportamento através do parâmetro `full_page_writes` no ficheiro de configuração. Com o LazyFS+, foram realizados dois testes distintos, um em que esta

proteção contra escritas parciais está ativa e outro em que está desativada. Foi possível, com a nossa solução, corromper uma base de dados quando o PostgreSQL não se protege contra este problema.

Para testar esta aplicação, foi utilizado o `pgbench` [106], um pequeno programa para correr testes no PostgreSQL. Numa primeira fase, executou-se o `pgbench` no PostgreSQL, sendo que a sua diretoria de dados estava numa diretoria onde o LazyFS+ estava montado. Esta fase de preparação é necessária para localizar escritas maiores que 4 KiB. O ficheiro alvo de faltas não deve ser o ficheiro de WAL, uma vez que este possui *checksums* e caso seja detetada alguma corrupção, esta será facilmente corrigida pelo PostgreSQL, pois as páginas em disco ainda não foram modificadas. Assim, o nosso alvo de estudo são os ficheiros que correspondem a páginas. Foi possível, com o *parser* do LazyFS+, detetar que a 100ª escrita numa página do PostgreSQL possui tamanho superior a 4 KiB. Deste modo, programou-se uma falta de atomicidade de escritas para um ficheiro que corresponde a uma página do PostgreSQL, para dividir essa escrita e persistir apenas a segunda porção.

Após a injeção da falta, aparentemente nenhum problema existe com a base de dados criada pelo `pgbench`. No entanto, para validar a existência, ou não, de erros/corrupção, utilizamos o `amcheck`, um módulo do PostgreSQL que disponibiliza funções que permitem verificar a coerência lógica da estrutura das relações, através da verificação de invariantes. A falha de um invariante pode levar a respostas erradas em interrogações SQL. O `amcheck` reporta o seguinte erro "*ERROR: heap tuple (27986,48) from table "pgbench_accounts"lacks matching index tuple within index "pgbench_accounts_pke""*. Isto significa que a falta injetada resulta numa corrupção silenciosa na base de dados.

Realizando a mesma experiência, mas com o comportamento de proteção contra escritas parciais de páginas ativado, o mesmo problema não acontece.

5.3 Discussão

O LazyFS+ permite testar sistemas de armazenamento relativamente à durabilidade dos dados e à sua *crash consistency*. De modo a ajudar os programadores na correção das aplicações, permite a reprodução de *bugs* e facilita a análise do fluxo de dados da aplicação (*i.e.*, as chamadas ao sistema executadas, os ficheiros manipulados, os dados que guardados em cache, entre outros).

No Capítulo 3 foram propostas um conjunto de questões às quais nos propusemos a responder. De seguida apresentamos uma resposta para cada uma delas.

- **Questão 1**. A maioria das aplicações analisadas utiliza `fsync` em determinados pontos da execução. No entanto, verifica-se que, em algumas situações, a sua utilização é evitada quando não o deveria ser. Dois dos casos mais preocupantes que foram descritos são o da inconsistência de

dados no ZooKeeper Ⓔ e o da perda de múltiplos pares chave-valor no LevelDB Ⓐ .

- **Questão 2**. O PostgreSQL é um exemplo de aplicação que se protege de forma rigorosa contra escritas parciais. Aliás, é das poucas aplicações que se preocupa em informar os utilizadores do nível de atomicidade que necessita para correto funcionamento.

- **Questão 3**. O LazyFS+ é a primeira ferramenta de injeção de faltas de reordenação e atomicidade de escritas que ajuda na tarefa de reprodução de *bugs* reportados pelos utilizadores. Assim, já não é necessário recorrer a alterações de código ou cortes da fonte de energia dos servidores para simular as falhas descritas. Com o LazyFS+ é inclusive possível **identificar a causa** de vários problemas, através da análise dos ficheiros com dados apenas em cache e da análise dos diagramas de operações das aplicações produzidos pelo *parser*. É também possível **confirmar a resolução** destes problemas.

  Este último ponto constitui um dos mais fortes para defender a utilidade do LazyFS+ e a necessidade de uma ferramenta para injeção de faltas e não apenas exploratória. Se a equipa de uma aplicação desejar verificar se um determinado *bug* foi corrigido, com o LazyFS+, basta reproduzir o teste que o desencadeia e, em alguns segundos, sabe se o problema foi ou não resolvido. Com uma ferramenta exploratória, seria necessário executá-la novamente sobre a aplicação para esta verificar se o número de vulnerabilidades diminuiu.

- **Questão 4**. Verifica-se que, frequentemente, as aplicações utilizam *checksums* para deteção de corrupções, mas são incapazes de as corrigir. Uma das conclusões retiradas deste estudo, e que, inclusive, já foi mencionada no trabalho de Ganesan *et al.*[55] é que **o comportamento mais comum perante a deteção de corrupções é falhar**. Obtém-se este mesmo resultado em *clusters* compostos por vários nodos e em situações em que só um deles falha, ou seja, mesmo existindo redundância, é possível que um nodo se torne indisponível devido a corrupção local.

- **Questão 5**. Nos casos de reordenação de escritas e escritas atómicas, a correção é mais complexa do que apenas a introdução de `fsyncs`. No caso da reordenação, por exemplo, verificou-se que o *bug* descoberto com a ajuda do LazyFS+ nas versões mais recentes do etcd não se resolve com `fsync` após cada escrita, como sugerido pela equipa do etcd. Já no caso Ⓖ , esta solução é suficiente.

  Para escrever atomicamente, pode utilizar-se a estratégia do ficheiro temporário e utilização do `rename`. Uma curiosidade é o facto de o ZooKeeper utilizar esta estratégia para o ficheiro que

indica a época atual do *cluster*, mas não a utilizar para o ficheiro de log, apesar de precisar que a primeira escrita para o ficheiro de log seja atómica. Por sua vez, o etcd possuía um problema de atomicidade numa versão antiga e utilizou esta mesma estratégia para o corrigir numa versão mais recente.

Como o problema de escritas parciais tem ganho maior notoriedade, as aplicações têm-se preparado de outras formas para tal. O Redis deteta os dados corrompidos e é capaz de os descartar, já o PostgreSQL implementa a técnica de escrever páginas completas após *checkpoints*.

Outro resultado interessante do estudo, que não foi abordado em estudos prévios, é o facto de que **algumas aplicações fornecem mensagens de erro enganadoras**. No caso do LevelDB, é reportada corrupção ao nível de um ficheiro que não está em nada relacionado com a causa do problema. No caso do ZooKeeper e do etcd, alguns *bugs* são causados pela existência de logs vazios, mas os erros apontam para o facto de ficheiros de *snapshot* não existirem, quando, na realidade, existem. O LazyFS+ é fundamental para identificar estes erros enganadores e apontar para as reais causas dos problemas.

Verificou-se também que várias aplicações utilizam logs transacionais de modo a permitir a recuperação após falhas, pois as operações são escritas para o log antes de serem aplicadas. No entanto, constatou-se que **algumas aplicações não se protegem contra corrupções nos logs**. O etcd, por exemplo, exige uma intervenção manual tediosa [88], enquanto que a solução mais automática do Zookeper não funciona em determinadas situações.

Capítulo 6
Conclusão

Acontecimentos como falhas de energia ou dos SOs ocorrem de forma inesperada e podem trazer consequências graves como indisponibilidade, inconsistência, perda e corrupção de dados. Estes problemas resultam, essencialmente, da necessidade de adotar estratégias para obter maior desempenho. A cache, por exemplo, embora contribue para uma melhoria de desempenho, traz desafios ao nível da durabilidade e ordenação dos dados. Nomeadamente, enquanto estiverem unicamente em cache, os dados estão suscetíveis a serem perdidos, e, para além disso, a ordem em que os dados são persistidos é potencialmente diferente da ordem em que foram atualizados. As aplicações devem, deste modo, assumir a responsabilidade de garantir a persistência e consistência dos seus dados. Contudo, isto não é uma tarefa trivial. Por um lado, nem sempre é intuitivo perceber em que situações a utilização do `fsync` (para persistir explicitamente os dados) é crítica e, por outro, nem sempre o `fsync` é suficiente, pois por vezes é necessário que certas escritas sejam realizadas atomicamente.

Muitas vezes os utilizadores deparam-se com situações decorrentes da falta ou incorreta aplicação de mecanismos de durabilidade e consistência dos dados por parte das aplicações. É comum estes problemas serem reportados nos respetivos repositórios ou fóruns. No entanto, verifica-se uma carência de meios para reproduzir estes problemas causados por falhas de energia. Múltiplos utilizadores sugerem alteração do código fonte da aplicação e o corte da fonte de energia do computador, de modo a reproduzir o problema com que se depararam. Outros utilizadores são incapazes de fornecer indicações de como reproduzir os *bugs*. Os próprios programadores das aplicações, por vezes, desconsideram os problemas reportados pelos utilizadores pela falta de meios de reprodução.

Esta dissertação propõe o LazyFS+, um SF que permite simular falhas de energia através da perda de dados em cache. A solução apresentada é capaz de simular a perda total de dados e adiciona uma camada de complexidade, permitindo simular a perda parcial. Assim o LazyFS+ permite, através da injeção de faltas baseada em *software*, simular falhas de energia e simular comportamentos dos SF, que levam à perda de uma parte dos dados. A solução proposta permite reproduzir de forma sistemática problemas encontrados em sistemas de armazenamento, pois possui uma cache interna que permite o

controlo total sobre os dados. Além disso, fornece *output* valioso para a sua compreensão.

Através de um estudo que inclui múltiplos *bugs* de quatro aplicações diferentes, reportados por utilizadores ou em artigos do trabalho relacionado, demonstramos como o LazyFS+ ajuda e facilita a reprodução e análise de problemas conhecidos. Com base neste estudo demonstramos também a capacidade da nossa solução de reproduzir *bugs* conhecidos em aplicações diferentes ou versões diferentes de uma aplicação. Analisando padrões de falhas e recorrendo às informações fornecidas por um *parser* dos logs do LazyFS+, também desenvolvido no âmbito desta dissertação, foi ainda possível encontrar com o LazyFS+ problemas desconhecidos em algumas aplicações.

Para além de reproduzir e ajudar a encontrar problemas, o LazyFS+ também pode ser utilizado na validação de mecanismos de recuperação ou proteção contra a perda parcial de dados. Deste modo, foram analisadas duas aplicações, o Redis e o PostgreSQL, de modo a verificar o correto funcionamento destes mecanismos.

6.1 Trabalho futuro

O LazyFS+ é uma ferramenta de injeção de faltas altamente extensível, pois, sendo um sistema de ficheiros que possui uma cache interna, permite o controlo total dos dados, possibilitando a introdução de um conjunto vasto de faltas relativas a dados (*e.g.* corrupção). Atualmente, o LazyFS+ está a ser utilizado pela equipa do etcd e os próprios têm sugerido adições para poderem integrar a nossa solução nas suas cadeias de testes. Uma das solicitações consiste em possibilitar a injeção de faltas de atomicidade e reordenação através de um FIFO [1] e outra consiste em proporcionar um novo tipo de injeção em que, durante um determinado período de tempo, as escritas para um dado ficheiro são descartadas [2].

Para além das adições para o etcd, existem outras possibilidades a estudar:

- **Faltas ao nível dos metadados**. O foco desta dissertação é a perda total ou parcial de dados, no entanto, a atualização dos metadados dos ficheiros é um processo que também pode ser vulnerável a falhas de energia. Por exemplo, existem SF cujos metadados podem ser atualizados antes dos dados, assim, após a falha de energia, o tamanho de um ficheiro pode ser superior à quantidade de dados que foi efetivamente escrita. Consequentemente, o ficheiro pode ficar com dados incorretos. Assim, seria interessante estender o LazyFS+ de modo a considerar também estes faltas ao nível dos metadados.

- **Injeção de faltas distribuída**. Alguns dos sistemas de armazenamento estudados são distri-

[1] *Allow injecting faults via a unix socket* - https://github.com/dsrhaslab/lazyfs/issues/3
[2] *drop_writes* - https://github.com/dsrhaslab/lazyfs/issues/3#issuecomment-1817471329

buídos, no entanto, com o LazyFS+ apenas foi possível injetar faltas locais. Os *bugs* encontrados durante o estudo tinham impacto apenas no nodo onde a falta é injetada. Seria interessante estudar casos em que a injeção de uma falta num nodo tivesse um impacto global.

Bibliografia

[1] George B. Finelli. «Characterization of Fault Recovery through Fault Injection on FTMP». Em: *IEEE Transactions on Reliability* R-36.2 (1987), pp. 164–170. DOI: 10.1109/TR.1987.5222333.

[2] J. Arlat, Y. Crouzet e J.-C. Laprie. «Fault injection for dependability validation of fault-tolerant computing systems». Em: *[1989] The Nineteenth International Symposium on Fault-Tolerant Computing. Digest of Papers*. 1989, pp. 348–355. DOI: 10.1109/FTCS.1989.105591.

[3] U. Gunneflo, J. Karlsson e J. Torin. «Evaluation of error detection schemes using fault injection by heavy-ion radiation». Em: *[1989] The Nineteenth International Symposium on Fault-Tolerant Computing. Digest of Papers*. 1989, pp. 340–347. DOI: 10.1109/FTCS.1989.105590.

[4] J.H. Barton *et al.* «Fault injection experiments using FIAT». Em: *IEEE Transactions on Computers* 39.4 (1990), pp. 575–582. DOI: 10.1109/12.54853.

[5] D. Avresky *et al.* «Fault injection for the formal testing of fault tolerance». Em: *[1992] Digest of Papers. FTCS-22: The Twenty-Second International Symposium on Fault-Tolerant Computing*. 1992, pp. 345–354. DOI: 10.1109/FTCS.1992.243566.

[6] Mendel Rosenblum e John K. Ousterhout. «The Design and Implementation of a Log-Structured File System». Em: *ACM Trans. Comput. Syst.* 10.1 (fev. de 1992), pp. 26–52. ISSN: 0734-2071. DOI: 10.1145/146941.146943. URL: https://doi.org/10.1145/146941.146943.

[7] Gregory R. Ganger e Yale N. Patt. «Metadata Update Performance in File Systems». Em: *First Symposium on Operating Systems Design and Implementation (OSDI 94)*. Monterey, CA: USENIX Association, nov. de 1994. URL: https://www.usenix.org/conference/osdi-94/metadata-update-performance-file-systems.

[8] Seungjae Han, K.G. Shin e H.A. Rosenberg. «DOCTOR: an integrated software fault injection environment for distributed real-time systems». Em: *Proceedings of 1995 IEEE International Computer Performance and Dependability Symposium*. 1995, pp. 204–213. DOI: 10.1109/IPDS.1995.395831.

[9] Timothy K. Tsai e Ravishankar K. Iyer. «FTAPE: A fault injection tool to measure fault Tolerance». English (US). Em: 1995, pp. 339–346. DOI: 10.2514/6.1995-1041.

[10] Mei-Chen Hsueh, T.K. Tsai e R.K. Iyer. «Fault injection techniques and tools». Em: *Computer* 30.4 (1997), pp. 75–82. DOI: 10.1109/2.585157.

[11] D.T. Stott *et al.* «NFTAPE: a framework for assessing dependability in distributed systems with lightweight fault injectors». Em: *Proceedings IEEE International Computer Performance and Dependability Symposium. IPDS 2000*. 2000, pp. 91–100. DOI: 10.1109/IPDS.2000.839467.

[12] Torres Wilfredo. *Software Fault Tolerance: A Tutorial*. Rel. téc. 2000.

[13] Alfredo Benso e Paolo Prinetto. *Fault Injection Techniques and Tools for Embedded Systems Reliability Evaluation*. Jan. de 2003.

[14] Diamantino Costa *et al.* «Xception™: A Software Implemented Fault Injection Tool». Em: *Fault Injection Techniques and Tools for Embedded Systems Reliability Evaluation*. Ed. por Alfredo Benso e Paolo Prinetto. Boston, MA: Springer US, 2003, pp. 125–139. ISBN: 978-0-306-48711-8. DOI: 10.1007/0-306-48711-X_8. URL: https://doi.org/10.1007/0-306-48711-X%5C%5F8.

[15] Tarek El-Ghazawi e Gideon Frieder. «Input-Output Operations». Em: *Encyclopedia of Computer Science*. GBR: John Wiley e Sons Ltd., 2003, pp. 874–879. ISBN: 0470864125.

[16] Livio B Soares, Orran Y Krieger e Dilma Da Silva. «Meta-Data Snapshotting: A Simple Mechanism for File System Consistency». Em: *Proceedings of the International Workshop on Storage Network Architecture and Parallel I/Os*. SNAPI '03. New Orleans, Louisiana: Association for Computing Machinery, 2003, pp. 41–52. ISBN: 9781450378215. DOI: 10.1145/1162618.1162624. URL: https://doi.org/10.1145/1162618.1162624.

[17] A. Avizienis *et al.* «Basic concepts and taxonomy of dependable and secure computing». Em: *IEEE Transactions on Dependable and Secure Computing* 1.1 (jan. de 2004), pp. 11–33. DOI: 10.1109/tdsc.2004.2. URL: https://doi.org/10.1109/tdsc.2004.2.

[18] A. Avizienis *et al.* «Basic concepts and taxonomy of dependable and secure computing». Em: *IEEE Transactions on Dependable and Secure Computing* 1.1 (jan. de 2004), pp. 11–33. DOI: 10.1109/tdsc.2004.2. URL: https://doi.org/10.1109/tdsc.2004.2.

[19] Haissam Ziade, Rafic Ayoubi e R. Velazco. «A Survey on Fault Injection Techniques». Em: *Int. Arab J. Inf. Technol.* 1 (jan. de 2004), pp. 171–186.

[20] Vijayan Prabhakaran *et al.* «IRON File Systems». Em: *SIGOPS Oper. Syst. Rev.* 39.5 (out. de 2005), pp. 206–220. ISSN: 0163-5980. DOI: 10.1145/1095809.1095830. URL: https://doi.org/10.1145/1095809.1095830.

[21] Vijayan Prabhakaran *et al.* «IRON File Systems». Em: *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*. SOSP '05. Brighton, United Kingdom: Association for Computing Machinery, 2005, pp. 206–220. ISBN: 1595930795. DOI: 10.1145/1095810.1095830. URL: https://doi.org/10.1145/1095810.1095830.

[22] Daniel P. Bovet e Marco Cesati. *Understanding the Linux Kernel*. 3ª ed. Sebastopol, CA: O'Reilly Media, 2006. ISBN: 9780596005658.

[23] Daniel P. Bovet e Marco Cesati. «Understanding the Linux Kernel». Em: 3ª ed. Sebastopol, CA: O'Reilly Media, 2006, p. 600. ISBN: 9780596005658.

[24] Daniel P. Bovet e Marco Cesati. «Understanding the Linux Kernel». Em: 3ª ed. Sebastopol, CA: O'Reilly Media, 2006, p. 32. ISBN: 9780596005658.

[25] Junfeng Yang, Can Sar e Dawson Engler. «EXPLODE: A Lightweight, General System for Finding Serious Storage System Errors». Em: *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI 06)*. Seattle, WA: USENIX Association, nov. de 2006. URL: https://www.usenix.org/conference/osdi-06/explode-lightweight-general-system-finding-serious-storage-system-errors.

[26] Israel Koren e C Mani Krishna. *Fault-Tolerant Systems*. Oxford, England: Morgan Kaufmann, mar. de 2007. ISBN: 9780120885251.

[27] Edmund B. Nightingale *et al.* «Rethink the Sync». Em: *ACM Trans. Comput. Syst.* 26.3 (set. de 2008). ISSN: 0734-2071. DOI: 10.1145/1394441.1394442. URL: https://doi.org/10.1145/1394441.1394442.

[28] *T-Mobile Sidekick Disaster: Danger's Servers Crashed, And They Don't Have A Backup*. Out. de 2009. URL: https://techcrunch.com/2009/10/10/t-mobile-sidekick-disaster-microsofts-servers-crashed-and-they-dont-have-a-backup/?guccounter=1%5C&guce%5C%5Freferrer=aHR0cHM6Ly9zdHJvbmdob2xkZGF0YS5jb20v%5C&guce%5C%5Freferrer%5C%5Fsig=AQAAANsBaePoz9dGhnsbbPVDwlT3SUS--M-2yeWBx4RaMDfZwt3bDTreoB5w%5C%5FJgHX8ENI99HLOStgchrHqVOqtZj3b4-I67X-myuZPU0u2sH4SE2mEr7zgiZobdtYSKW7V-ICNtEIZfhPH7H7l27r253WcYOccHGDDOX9EQweGltdeGu.

[29] Robert Love. *Linux Kernel Development*. 3rd. Addison-Wesley Professional, 2010. ISBN: 0672329468.

[30] Haryadi S. Gunawi *et al.* «FATE and DESTINI: A Framework for Cloud Recovery Testing». Em: NSDI'11. Boston, MA: USENIX Association, 2011, pp. 238–252.

[31] Jan Vitek e Tomas Kalibera. «Repeatability, Reproducibility, and Rigor in Systems Research». Em: *Proceedings of the Ninth ACM International Conference on Embedded Software*. EMSOFT '11. Taipei, Taiwan: Association for Computing Machinery, 2011, pp. 33–38. ISBN: 9781450307147. DOI: 10.1145/2038642.2038650. URL: https://doi.org/10.1145/2038642.2038650.

[32] Abraham Silberschatz, Peter B Galvin e Greg Gagne. *Operating System Concepts*. Nashville, TN: John Wiley & Sons, dez. de 2012.

[33] Vijay Chidambaram *et al.* «Optimistic Crash Consistency». Em: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. SOSP '13. Farminton, Pennsylvania: Association for Computing Machinery, 2013, pp. 228–243. ISBN: 9781450323888. DOI: 10.1145/2517349.2522726. URL: https://doi.org/10.1145/2517349.2522726.

[34] Vijay Chidambaram *et al.* «Optimistic Crash Consistency». Em: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. SOSP '13. Farminton, Pennsylvania: Association for Computing Machinery, 2013, pp. 228–243. ISBN: 9781450323888. DOI: 10.1145/2517349.2522726. URL: https://doi.org/10.1145/2517349.2522726.

[35] Nicholas Hunt *et al.* «DDOS: Taming Nondeterminism in Distributed Systems». Em: *SIGPLAN Not.* 48.4 (mar. de 2013), pp. 499–508. ISSN: 0362-1340. DOI: 10.1145/2499368.2451170. URL: https://doi.org/10.1145/2499368.2451170.

[36] Mai Zheng *et al.* «Understanding the Robustness of SSDs under Power Fault». Em: *11th USENIX Conference on File and Storage Technologies (FAST 13)*. San Jose, CA: USENIX Association, fev. de 2013, pp. 271–284. ISBN: 978-1-931971-99-7. URL: https://www.usenix.org/conference/fast13/technical-sessions/presentation/zheng.

[37] Roozbeh Bakhshi, Surya Kunche e Michael Pecht. «Intermittent Failures in Hardware and Software». Em: *Journal of Electronic Packaging* 136.1 (fev. de 2014). DOI: 10.1115/1.4026639. URL: https://doi.org/10.1115/1.4026639.

[38] Haryadi Gunawi *et al.* «What Bugs Live in the Cloud?» Em: nov. de 2014, pp. 1–14. DOI: 10.1145/2670979.2670986.

[39] Thanumalayan Sankaranarayana Pillai *et al.* «All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-Consistent Applications». Em: *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. Broomfield, CO: USENIX Association, out. de 2014, pp. 433–448. ISBN: 978-1-931971-16-4. URL: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/pillai.

[40] Mai Zheng *et al.* «Torturing Databases for Fun and Profit». Em: *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. Broomfield, CO: USENIX Association, out. de 2014, pp. 449–464. ISBN: 978-1-931971-16-4. URL: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/zheng%5C%5Fmai.

[41] *Airbus A400M plane crash linked to software fault*. Mai. de 2015. URL: https://www.bbc.com/news/technology-32810273.

[42] Thanumalayan Sankaranarayana Pillai *et al.* «Crash Consistency: Rethinking the Fundamental Abstractions of the File System». Em: *Queue* 13.7 (jul. de 2015), pp. 20–28. ISSN: 1542-7730. DOI: 10.1145/2800695.2801719. URL: https://doi.org/10.1145/2800695.2801719.

[43] *Advantages and Disadvantages of UPS and Generators: For the Load shedding Stricken Nation*. Jul. de 2016. URL: https://www.linkedin.com/pulse/advantages-disadvantages-ups-generators-load-shedding-sultan-zafar/.

[44] Ramnatthan Alagappan *et al.* «Correlated Crash Vulnerabilities». Em: *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*. OSDI'16. Savannah, GA, USA: USENIX Association, 2016, pp. 151–167. ISBN: 9781931971331.

[45] James Bornholt *et al.* «Specifying and Checking File System Crash-Consistency Models». Em: *SIGPLAN Not.* 51.4 (mar. de 2016), pp. 83–98. ISSN: 0362-1340. DOI: 10.1145/2954679.2872406. URL: https://doi.org/10.1145/2954679.2872406.

[46] James Bornholt *et al.* «Specifying and Checking File System Crash-Consistency Models». Em: *SIGARCH Comput. Archit. News* 44.2 (mar. de 2016), pp. 83–98. ISSN: 0163-5964. DOI: 10.1145/2980024.2872406. URL: https://doi.org/10.1145/2980024.2872406.

[47] James Bornholt *et al.* «Specifying and Checking File System Crash-Consistency Models». Em: *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '16. Atlanta, Georgia, USA: Association for Computing Machinery, 2016, pp. 83–98. ISBN: 9781450340915. DOI: 10.1145/2872362.2872406. URL: https://doi.org/10.1145/2872362.2872406.

[48] *CharybdeFS: a new fault-injecting filesystem for software testing*. Jun. de 2016. URL: https://www.scylladb.com/2016/02/16/fault-injection-filesystem-software-testing/.

[49] Yanyan Jiang *et al.* «Crash Consistency Validation Made Easy». Em: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE 2016. Seattle, WA, USA: Association for Computing Machinery, 2016, pp. 133–143. ISBN: 9781450342186. DOI: 10.1145/2950290.2950327. URL: https://doi.org/10.1145/2950290.2950327.

[50] Eric Koskinen e Junfeng Yang. «Reducing Crash Recoverability to Reachability». Em: *SIGPLAN Not.* 51.1 (jan. de 2016), pp. 97–108. ISSN: 0362-1340. DOI: 10.1145/2914770.2837648. URL: https://doi.org/10.1145/2914770.2837648.

[51] Eric Koskinen e Junfeng Yang. «Reducing Crash Recoverability to Reachability». Em: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '16. St. Petersburg, FL, USA: Association for Computing Machinery, 2016, pp. 97–108. ISBN: 9781450335492. DOI: 10.1145/2837614.2837648. URL: https://doi.org/10.1145/2837614.2837648.

[52] Roberto Natella, Domenico Cotroneo e Henrique S. Madeira. «Assessing Dependability with Software Fault Injection: A Survey». Em: *ACM Comput. Surv.* 48.3 (fev. de 2016). ISSN: 0360-0300. DOI: 10.1145/2841425. URL: https://doi.org/10.1145/2841425.

[53] Roberto Natella, Domenico Cotroneo e Henrique S. Madeira. «Assessing Dependability with Software Fault Injection: A Survey». Em: *ACM Comput. Surv.* 48.3 (fev. de 2016). ISSN: 0360-0300. DOI: 10.1145/2841425. URL: https://doi.org/10.1145/2841425.

[54] Helgi Sigurbjarnarson *et al.* «Push-Button Verification of File Systems via Crash Refinement». Em: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. Savannah, GA: USENIX Association, nov. de 2016, pp. 1–16. ISBN: 978-1-931971-33-1. URL: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/sigurbjarnarson.

[55] Aishwarya Ganesan *et al.* «Redundancy Does Not Imply Fault Tolerance: Analysis of Distributed Storage Reactions to Single Errors and Corruptions». Em: FAST'17. Santa clara, CA, USA: USENIX Association, 2017, pp. 149–165. ISBN: 9781931971362.

[56] Ashlie Martinez e Vijay Chidambaram. «CrashMonkey: A Framework to Automatically Test File-System Crash Consistency». Em: *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 17)*. Santa Clara, CA: USENIX Association, jul. de 2017. URL: https://www.usenix.org/conference/hotstorage17/program/presentation/martinez.

[57] Thanumalayan Sankaranarayana Pillai *et al.* «Application Crash Consistency and Performance with CCFS». Em: *ACM Trans. Storage* 13.3 (set. de 2017). ISSN: 1553-3077. DOI: 10.1145/3119897. URL: https://doi.org/10.1145/3119897.

[58] Remzi H. Arpaci-Dusseau e Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. 1.00. Arpaci-Dusseau Books, ago. de 2018.

[59] *Boeing issues warning on potential instrument malfunction after Indonesia crash*. Nov. de 2018. URL: https://www.washingtonpost.com/world/asia%5C%5Fpacific/boeing-issues-warning-on-potential-instrument-malfunction-after-indonesia-crash/2018/11/07/b43168b6-e265-11e8-a1c9-6afe99dddd92%5C%5Fstory.html.

[60] Jayashree Mohan *et al.* «Finding Crash-Consistency Bugs with Bounded Black-Box Crash Testing». Em: *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*. OSDI'18. Carlsbad, CA, USA: USENIX Association, 2018, pp. 33–50. ISBN: 9781931971478.

[61] Miguel Amaral *et al.* «FaultSee: Reproducible Fault Injection in Distributed Systems». Em: *2020 16th European Dependable Computing Conference (EDCC)*. 2020, pp. 25–32. DOI: 10.1109/EDCC51268.2020.00014.

[62] Dong Hyun Kang *et al.* «Making Application-Level Crash Consistency Practical on Flash Storage». Em: *IEEE Transactions on Parallel and Distributed Systems* 31.5 (2020), pp. 1009–1020. DOI: 10.1109/TPDS.2019.2959305.

[63] Anthony Rebello *et al.* «Can Applications Recover from fsync Failures?» Em: *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, jul. de 2020, pp. 753–767. ISBN: 978-1-939133-14-4. URL: https://www.usenix.org/conference/atc20/presentation/rebello.

[64] *Advantages and Disadvantages of UPS*. Ago. de 2021. URL: https://www.chtips.com/computer-fundamentals/advantages-and-disadvantages-of-ups/.

[65] *More details about the October 4 outage*. Out. de 2021. URL: https://engineering.fb.com/2021/10/05/networking-traffic/outage-details/.

[66] *Using named pipes*. Mar. de 2021. URL: https://www.ibm.com/docs/en/zos/2.2.0?topic=io-using-named-pipes.

[67] Ram Alagappan e Peter Alvaro. «Crash Consistency: Keeping Data Safe in the Presence of Crashes is a Fundamental Problem.» Em: *Queue* 20.4 (set. de 2022), pp. 107–115. ISSN: 1542-7730. DOI: 10.1145/3561654. URL: https://doi.org/10.1145/3561654.

[68] João Pedro Rodrigues Azevedo. *LazyFS: A file system for assessing applications data durability*. 2022.

[69] Daejun Park e Dongkun Shin. «Stackable Transactional File System Using Kernel-Level WAL». Em: *IEEE Access* 10 (2022), pp. 110088–110099. DOI: 10.1109/ACCESS.2022.3214521.

[70] *Random Seeds and Reproducibility*. Mai. de 2022. URL: https://towardsdatascience.com/random-seeds-and-reproducibility-933da79446e3.

[71] *TCP/IP Checksum Testing*. Set. de 2022. URL: https://www.ibm.com/docs/en/zvm/7.2?topic=verification-tcpip-checksum-testing.

[72] *Volume of data/information created, captured, copied, and consumed worldwide from 2010 to 2020, with forecasts from 2021 to 2025*. Set. de 2022. URL: https://www.statista.com/statistics/871513/worldwide-data-created/.

[73] Jacob Van Geffen *et al.* «Synthesis-aided crash consistency for storage systems». Em: *ECOOP 2023*. 2023. URL: https://www.amazon.science/publications/synthesis-aided-crash-consistency-for-storage-systems.

[74] Hayley LeBlanc *et al.* «Chipmunk: Investigating Crash-Consistency in Persistent-Memory File Systems». Em: *Proceedings of the Eighteenth European Conference on Computer Systems*. EuroSys '23. Rome, Italy: Association for Computing Machinery, 2023, pp. 718–733. ISBN: 9781450394871. DOI: 10.1145/3552326.3567498. URL: https://doi.org/10.1145/3552326.3567498.

[75] Xiaojian Liao *et al.* «Efficient Crash Consistency for NVMe over PCIe and RDMA». Em: *ACM Trans. Storage* 19.1 (jan. de 2023). ISSN: 1553-3077. DOI: 10.1145/3568428. URL: https://doi.org/10.1145/3568428.

[76] *A series of tools for ZooKeeper*. URL: https://zookeeper.apache.org/doc/current/zookeeperTools.html.

[77] Apache. *Apache Cassandra*. URL: https://cassandra.apache.org/%5C%5F/index.html.

[78] Apache. *Apache Kafka*. URL: https://kafka.apache.org/.

[79] Apache. *Apache ZooKeeper*. URL: https://zookeeper.apache.org/.

[80] *Are there any DBMSs that manage disk blocks directly?* URL: https://www.quora.com/Are-there-any-DBMSs-that-manage-disk-blocks-directly.

[81] *Big Data : What it is and why it matters*. URL: https://www.sas.com/en%5C%5Fus/insights/analytics/big-data-analytics.html.

[82] *BitTorrent*. URL: https://www.bittorrent.com/pt.

[83] César Borges e João Paulo. *Realistic Assessment of Faults in Storage Systems*. URL: https://dsr-haslab.github.io/repository/bp21.pdf.

[84] Sergey Bronnikov. *UnreliableFS: A FUSE-based fault injection filesystem*. URL: https://github.com/ligurio/unreliablefs.

[85] *Chaos Monkey*. URL: https://netflix.github.io/chaosmonkey/.

[86] *Chapter 22. Write Barriers*. URL: https://access.redhat.com/documentation/en-us/red%5C%5Fhat%5C%5Fenterprise%5C%5Flinux/6/html/storage%5C%5Fadministration%5C%5Fguide/ch-writebarriers.

[87] *Checksums and Their Use in ZFS*. URL: https://openzfs.github.io/openzfs-docs/Basic%5C%20Concepts/Checksums.html.

[88] *Data Corruption*. URL: https://etcd.io/docs/v3.6/op-guide/data%5C%5Fcorruption/.

[89] *Data Growth*. URL: https://www.idera.com/glossary/data-growth/.

[90] *Data inconsistency caused by DIFF sync uncommitted log*. URL: https://issues.apache.org/jira/browse/ZOOKEEPER-3911?jql=project%5C%20%5C%3D%5C%20ZOOKEEPER%5C%20AND%5C%20text%5C%20~%5C%20inconsistency.

[91] *Ensuring data reaches disk*. URL: https://lwn.net/Articles/457667/.

[92] *ETCD data gets corrupted with error "walpb: crc mismatch"*. URL: https://github.com/etcd-io/etcd/issues/11488%5C#%5C#issuecomment-570241755.

[93] *Filebench - A Model Based File System Workload Generator*. URL: https://github.com/filebench/filebench.

[94] *Flushing out pdflush*. URL: https://lwn.net/Articles/326552/.

[95] *Full page writes*. URL: https://wiki.postgresql.org/wiki/Full%5C%5Fpage%5C%5Fwrites.

[96] *Functions, types, and advantages of uninterruptible power supply (UPS)*. URL: https://www.knowcomputing.com/functions-types-and-advantages-of-uninterruptible-power-supply/%5C#disadvantages-ups.

[97] *FUSE: The reference implementation of the Linux FUSE (Filesystem in Userspace) interface*. URL: https://github.com/libfuse/libfuse.

[98] *Google Cloud Status Dashboard*. URL: https://status.cloud.google.com/incident/compute/15056.

[99] Apache Hadoop. *HDFS Architecture Guide*. URL: https://hadoop.apache.org/docs/r1.2.1/hdfs%5C%5Fdesign.html.

[100] Kyle Kingsbury. *Jepsen: Distributed Systems Safety Research*. URL: https://jepsen.io/.

[101] *LazyFS: A FUSE Filesystem with an internal dedicated page cache, which can be used to simulate data loss on unsynced writes*. URL: https://github.com/dsrhaslab/lazyfs.

[102] MongoDB. *MongoDB*. URL: https://www.mongodb.com/.

[103] *open(2) – Linux manual page*. URL: https://man7.org/linux/man-pages/man2/open.2.html.

[104] *Persistent Memory is Revolutionary*. URL: https://pmem.io/learn/.

[105] *PetardFS - a FUSE filessytem for injecting intentional errors*. URL: https://github.com/jrandall/petardfs.

[106] *pgbench*. URL: https://www.postgresql.org/docs/current/pgbench.html.

[107] *PostgreSQL*. URL: https://www.postgresql.org/.

[108] PostgreSQL. *Checkpoint*. URL: https://www.postgresql.org/docs/current/sql-checkpoint.html.

[109] *Potential data inconsistency due to the inconsistency between ZKDatabase.committedLog and dataTree in Trunc sync*. URL: https://issues.apache.org/jira/browse/ZOOKEEPER-3607?jql=project%5C%20%5C%3D%5C%20ZOOKEEPER%5C%20AND%5C%20text%5C%20~%5C%20inconsistency.

[110] *Potential ephemeral nodes inconsistent due to global session inconsistent with fuzzy snapshot*. URL: https://issues.apache.org/jira/browse/ZOOKEEPER-3144?jql=project%5C%20%5C%3D%5C%20ZOOKEEPER%5C%20AND%5C%20text%5C%20~%5C%20inconsistency.

[111] *Principles of Chaos Engineering*. URL: https://principlesofchaos.org/.

[112] *Redis*. URL: https://redis.io/.

[113] *Reliability vs Availability: What's The Difference?* URL: https://www.bmc.com/blogs/reliability-vs-availability/.

[114] *rename(2) – Linux manual page*. URL: https://man7.org/linux/man-pages/man2/rename.2.html.

[115] Casey Rosenthal *et al. Why Do Chaos Engineering?* URL: https://www.oreilly.com/library/view/chaos-engineering/9781491988459/ch01.html.

[116] *ScyllaDB*. URL: https://www.scylladb.com/.

[117] *std::atomic*. URL: https://en.cppreference.com/w/cpp/atomic/atomic.

[118] *The Raft Consensus Algorithm*. URL: https://raft.github.io/.

[119] *The snapshot is not saved to disk and may cause data inconsistency*. URL: https://issues.apache.org/jira/browse/ZOOKEEPER-3220?jql=project%5C%20%5C%3D%5C%20ZOOKEEPER%5C%20AND%5C%20text%5C%20~%5C%20%5C%22fsync%5C%22.

[120] Theodore Ts'o. *fsck(8) - Linux man page*. URL: https://linux.die.net/man/8/fsck.

[121] *Use containers to Build, Share and Run your applications*. URL: https://www.docker.com/resources/what-container/.

[122] *ZooKeeper Programmer's Guide*. URL: https://zookeeper.apache.org/doc/current/zookeeperProgrammers.html.

[123] *ZOOKEEPER-2872: Interrupted snapshot sync causes data loss*. URL: https://github.com/apache/zookeeper/pull/333.